VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Webový editor pro podporu víceuživatelské online práce

Web based editor supporting online-multiuser cooperation

2014 Martin Lonský

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Martin Lonský**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Webový editor pro podporu víceuživatelské online práce
Web Based Editor Supporting Online-multiuser Cooperation

Zásady pro vypracování:

Cílem práce je zmapovat a popsat současné možnosti víceuživatelské kolaborativní práce na webu, a to s využitím moderních skriptovacích přístupů.

1. Seznamte se a detailně nastudujte vývojové prostředí Node.js a srovnejte jej s dalšími prostředími umožňující asynchroní implementaci v prostředí webu.
2. Definujte požadavky pro aplikaci umožňující editaci textového obsahu v rámci víceuživatelského přístupu v "reálném čase". Součástí bude rovněž řešení pro sdílení souborů v rámci tohoto režimu.
3. Analyzujte a navrhněte odpovídající řešení, specificky s ohledem na asynchronní přístup a kooperaci uživatelů.
4. Implementujte tuto aplikaci v Node.js a využijte možností zvolné nonsql databáze.
5. Zhodnoťte výslednou aplikaci, a to včetně uživatelského testování a měření výkonosti.

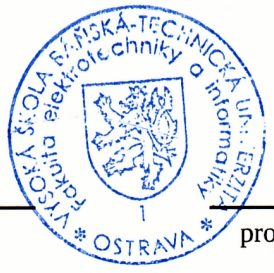
Seznam doporučené odborné literatury:

[1] Manuel Kiessling: The Node Beginner Book. Leanpub. 2011. ISBN: 1471628442
[2] David Herron: Node Web Development. Packt Publishing. 2011
[3] Kristina Chodorow, Michael Dirolf: MongoDB: The Definitive Guide. O'Reilly Media. 2010

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Michal Radecký, Ph.D.**

Datum zadání: 01.09.2013
Datum odevzdání: 07.05.2014



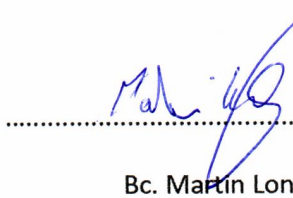
doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Čestné prohlášení:

Prohlašuji, že jsem tuto bakalářskou/diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

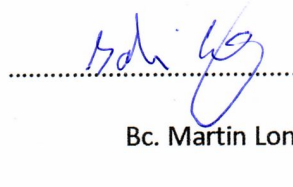
Ve Frýdku-Místku dne 17.4.2014



.....

Bc. Martin Lonský

Souhlasím se zveřejněním této bakalářské/diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v bakalářských/magisterských programech VŠB-TU Ostrava.



.....

Bc. Martin Lonský

Na tomto místě bych rád poděkoval lidem, kteří mi s diplomovou práci pomáhali. Panu Ing. Michalu Radeckému, Ph.D. za průběžné konzultace,
Mgr. Danielu Bielczykovi a Ing. arch. Petru Bílému za grafické zpracování prototypu uživatelského rozhraní a Bc. Tomášovi Přečkovi za návrh loga aplikace.

Abstrakt

V diplomové práci se zabývám problematikou přístupu k souborům pomocí webového rozhraní. Toto webové rozhraní umožňuje jednoduchou správu souborů a jejich víceuživatelskou editaci v reálném čase. Výsledkem je real-time víceuživatelský editor, zobrazující editovaný soubor se zvýrazněnou syntaxí podle jeho typu, reagující na události jiných uživatelů nebo dalších otevřených oken tak, aby byli všichni uživatelé synchronizováni. Součástí editoru je také online chat, který jako chatroom využívá právě otevřených souborů. Využil jsem možností, které nabízí spojení technologií NodeJS, Socket.io a HTML5.

Klíčová slova: Víceuživatelský editor, real-time, NodeJS, Socket.IO, HTML5

Abstract

This thesis deals with the issue of access to files through the web interface. This web interface allows you to manage files easily with real-time mlti-user cooperation. The result is editor with syntax highlighting according to its type reacting to events other users or windows. All users are synchronized. One part of the editor is online chat that uses the currently opened file as chatroom. I took advantage of the possibilities offered by the combination od technologies NodeJS, Socket.IO and HTML5.

Keywords: Multi-user editor, real-time, NodeJS, Socket.IO, HTML5

Seznam použitých zkratek a symbolů

IE	–	Internet Explorer
FTP	–	File Transport Protocol
SVN	–	Subversion
HTML	–	Hyper Text Markup Language
AJAX	–	Asynchronous JavaScript and XML
DOM	–	Document Object Model
CSS	–	Cascading Style Sheets
JS	–	JavaScript
PHP	–	Hypertext Preprocesso
TCP	–	Transmission Control Protocol
HTTP	–	Hypertext Transfer Protocol
ECMA	–	European Computer Manufacturers Association
URI	–	Uniform Resource Identifier
JSON	–	JavaScript Object Notation
BSON	–	Binary JSON
SQL	–	Structured Query Language
NPM	–	Node Package Manager
UI	–	User Interface
WS	–	WebSocket
DB	–	Data Base
SŘBD	–	Systém Řízení Báze Dat

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Ve své bakalářské práci jsem se okrajově zmínil o nově se rozvíjejících technologiích a platformě NodeJS. Rozhodl jsem se tedy více se zaměřit na způsoby vývoje internetových aplikací současné doby a využít tak potenciálu internetových prohlížečů. Také jsem se, z důvodu neudržitelnosti zpětné kompatibility, rozhodl, že nebudu ve svém nástroji řešit problematiku funkcionality některých částí kódů ve starších prohlížečích, zejména pak starší řady prohlížečů Microsoft IE. Motivací k vytvoření online editoru pro mě byla současná situace v práci. Doposud využíváme vlastní implementaci FTP, které je ovšem nestabilní a nemá jej kdo opravit. V neposlední řadě mě k nástroji přiměla myšlenka synchronizované práce. Často při vývoji nastane okamžik, kdy je zapotřebí součinnost dalšího člověka. Ať už jenom žádost o radu, nebo samotná práce na části zdrojového kódu. V případě FTP se tato součinnost nedá bez využití dalších nástrojů[1] realizovat. Kdokoliv by mohl namítnout, že tyto problémy přece řeší verzovací systémy jako jsou například GIT nebo SVN. Mé rozhodnutí bylo značnou měrou ovlivněno složitostí obsluhy verzovacích systémů. „Commitnout“ nějaké změny na server dokáže asi každý. Problém ale nastává v situaci, kdy dojde k nějaké kolizi. Řešení těchto problémů je nad rámec znalostí kodérů[2], kterým tato aplikace má primárně sloužit. Cílem je také samozřejmě snížení počtu obsluhovaných nástrojů na jeden, který dokáže uspokojit všechny potřeby s kódováním internetových aplikací, vyřeší kolize automaticky a udrží synchronizovanou verzi dokumentu v reálném čase na všech stanicích.

[1] Nástroje umožňující synchronizaci a ideálně grafické zobrazení rozdílů verzí
[2] Vývojář v HTML, CSS a JS. Typicky napojuje grafický návrh na již hotový IS.

2 HTML5

HTML5 je poslední standard pro HTML, který vyšel v roce 2009. Předešlé verze HTML přišly již v roce 1999(HTML 4.01) a radikálně tím změnily internet. HTML5 bylo navrženo tak, aby nahradilo všechny předešlé verze HTML. Byl kladen důraz na to, aby web poskytoval bohatý obsah bez nutnosti využití dalších pluginů například v podobě AdobeFlash. HTML5 umožňuje tvorbu složitých aplikací na straně klienta a je(mělo by být) kompatibilní skrze všechny platformy. Dnes již převážná většina internetových prohlížečů HTML5 nativně podporuje. [16] HTML5 také rozšiřuje formulářové prvky o některá další pole, jako je například placeholder(zobrazení libovolného textu, pokud je položka prázdná), required(povinná položka), nebo pattern(nastavení regulárního výrazu `RegExp` pro validaci dat položky).

HTML5 vystihuje nálsedující formule:

$$\text{HTML5} \sim= HTML + CSS + JS \text{ [22]}$$

Předchozí formule říká, že HTML5 přibližně znamená mix HTML, CSS a JS. Tyto technologie dohromady umožňují vývojářům tvořit webové aplikace, jako by byly tvořeny jako desktopové aplikace.

Co HTML5 například podporuje je v tabulce 1:

Název	Využito v editoru
Přehrávání multimédií	Ano
Grafika	Ano
XHTMLHttpRequest 2	Ano
FileAccess	Ano
WevSocket	Ano
Nová sémantika	Ano
CSS3	Ano

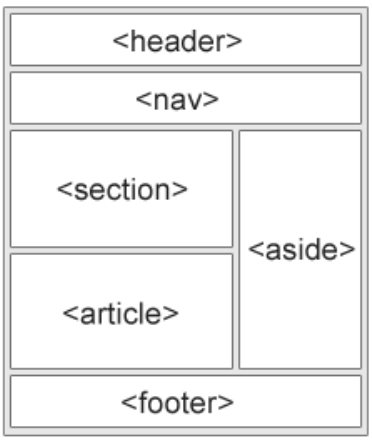
Tabulka 1: HTML5 možnosti

2.1 Sémantický web

HTML5 přidává nové elementy do DOM dokuemntu nové položky `header`, `nav`, `article`, `section`, `aside`, `footer` atd., které se používají místo dřívějšího řešení pomocí DIVů. Tyto elementy přímo říkají, k jakému účelu jednotlivé části kódu slouží a definují jejich význam na webu[3]. Grafický náhled využití nových tagů je na obrázku 1.

Sémantický web také značně ulehčuje analýzu webových stránek vyhledávačům.

[3]Tyto položky jsou samozřejmě také přístupné ke stylování v CSS3, stejně jako div, nebo span.



Obrázek 1: Sémantický web HTML5 [16]

2.2 WebSocket

WebSocket(dále jen „WS“) je protokol založený na TCP, který umožňuje fullduplex komunikaci na jediném spojení. Využívá k přenosu a handshake(Upgrade request) protokolu HTTP s tím, že spojení zůstává otevřené po celou dobu spojení.

WS byl vytvořen pro internetové prohlížeče a servery a umožňuje komunikaci mezi klientem a serverem v reálném čase, což se využívá, mimo jiné, ve hrách i aplikacích, kde je třeba zajistit stálé a rychlé spojení se serverem.

Následující ukázky obsahují hlavičku handshake požadavku od klienta1 a serveru2[18].

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

Výpis 1: WebSocket Client handshake

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

Výpis 2: WebSocket Server handshake

Handshake probídá tak, že klient zasílá svůj náhodný klíč serveru v položce hlavičky `Sec-WebSocket-Key`. Server k přijaté hodnotě přidá string

`258EAFA5-E914-47DA-95CA-C5AB0DC85B11`

a vytvoří hash pommocí kryptovacího mechanismu SHA-1. Výsledek převede pomocí `base64` kryptování a zašle zpět klientovi jako `Sec-WebSocket-Accept` s kódem 101 (Switching Protocols). Cokoliv jiného, než status 101 znamená, že handshake neproběhl.

Specifikace WS umožňuje dvě URI schémata:

ws-URI: `ws://host[:port]path[?query]`
wss-URI: `wss://host[:port]path[?query]`

Výchozí číslo portu je port 80 pro ws a 443 pro wss. wss značí, že jde, jak už číslo portu napovídá, o šifrované spojení.

Server typicky dokáže komunikovat s mnoha otevřenými WS. Tomuto se budeme blíže věnovat v kapitole Socket.IO (4.5).

2.2.1 WebSocket v JS - client

V moderních prohlížečích je nativně JS funkce `WebSocket` pro práci s WS. Jako parametr přijímá adresu WS serveru. Dále se nastavují k prototypu funkce, které jsou volány při událostech `onopen`, `onmessage` a `onclose` viz. ukázka 3.

```
var ws = new WebSocket('ws://example.com:3000');
ws.onopen = function(){
    console.log('connected');
}
ws.onmessage = function(e){
    console.log(e.data);
}
```

Výpis 3: WebSocket v JS - client

2.3 CSS3

CSS3 je poslední standard CSS a umožňuje řadu novinek pro webové aplikace:

- **Nové selektory** - HTML5 sémantika
- **Animace**(timeline, easing) a **grafické efekty**(gradients)
  Nyní je možné navázat nějakou plynulou změnu a navázat nějaké reakce na to, že animace byla dokončena apod. Mnohé věci již není třeba řešit přes JS, ale dají se vyřešit už v samotném CSS.
- **2D/3D Transformace**
- **Responsive design**
  Na úrovni CSS se řeší problémy s velikostí viditelné části okna a reakce na změnu velikosti okna. Pro jednotlivé elementy je možné, na předdefinovaných hranicích velikosti okna, měnit jejich CSS atributy a měnit jejich vzhled i umístění nebo je úplně skrýt/nahradit. Responsive design je úzce spjat s tvorbou internetových aplikací pro mobilní telefony, tablety a zároveň pro desktopy. Není třeba již tvořit zvlášť mobilní verze aplikace a přesměrovávat na ně, pokud je rozpoznáno mobilní zařízení. Vše vyřeší CSS samo.
  Responsive design se deklaruje pomocí `media query`4

```
@media screen and (max-width: 980px) { //do sirky 980px bude text HTML5 kontejneru main bily
    main {
        color: #fff; \\zkraceny zapis barvy
    }
}
```

Výpis 4: Responsive design query

CSS3 má také i strukturální omezení. Některé deklarace musí být striktně na začátku dokumentu, jinak nebudou fungovat. Jedná se hlavně o `@import`. Není možné importovat css soubory uprostřed souboru. Prohlížeč nás na to nijak neupozorní, v síťové analýze je také vidět, že se styly stahují, ale na webu již nedojde k jejich aplikaci.

3 AJAX

V dnešní době ještě stále většina aplikací využívá AJAX pro jednodušší a rychlejší práci s webovým obsahem. AJAX stejně jako následně modernější WebSocket(2.2) může komunikovat se serverem a načítat data na pozadí a odstiňuje tak uživatele od nutnosti obnovit stránku, aby se změny projevily.

AJAX je postaven na `XMLHttpRequest`. Vytváří standardní HTTP požadavek. Můžeme si také zvolit způsob, jakým chceme požadavek vytvořit: synchronní nebo asynchronní.

Komunikace se serverem:

1. Klient otevírá spojení se serverem.
2. Klient přenáší data z hlavičky a případně přenáší data pomocí POST.
3. Klient čeká, až server zpracuje požadavek.
4. Server zasílá odpověď a nastavuje hlavičky odpovědi.
5. Spojení je uzavřeno.

AJAX se dnes nejčastěji používá pomocí knihovny `jQuery`, která zastřešuje JS skrze všechny platformy a typy internetových prohlížečů. Je to kompaktní framework, který sjednocuje chování a názvosloví funkcí jednotlivých prohlížečů do jediné rozšiřitelné knihovny. Využití AJAXu je zde velmi jednoduché. Pokud například chceme, aby nám server odpověděl na požadavek zaslaný metodou GET, pak jednoduše implementujeme:

```
jQuery.post('/', {data: jQuery('body').html()}, function(data){
    console.log(data.state);
}, 'json').error(function(e){
    console.error(e);
}).always(function(){
    console.log('Called everytime.');
});
```

Výpis 5: jQuery AJAX POST

V ukázce 5 voláme server s adresou aktuální webové stránky s URI / a předáváme metodou POST celý obsah HTML elementu body. Data, která nám zpět vrátí server očekáváme v JSON. Pokud vznikne na serveru chyba nebo data nejsou ve formátu JSON, potom se vyvolá část kódu error. jQuery umožňuje v části always vyvolat událost, která se provede vždy, když server zašle odpověď.

jQuery lze také rozšířit o jQuery UI, které nabízí již hotová řešení některých prvků uživatelského rozhraní, jako je například `DatePicker`(výběr data z kalendáře)[4].

[4]jQuery také obsahuje množství pluginů a možnost tvořit vlastní pluginy.

4 Platforma NodeJS

4.1 Google v8

Google v8 je open-source JavaScript engine implementovaný v C++, který převádí JS kód do strojového jazyka.[8] Využívá ECMAScript který je standardizován v ECMA-262. ECMAScript je standardizovaný JavaScript neziskovou standardizační společností Ecma International.[9]. Celý mrak ECMAScript je v příloze na obrázku 22

Motivací Googlu pro vytvoření překladače JS kódu do strojového jazyka bylo několikanásobné zrychlení zpracování programu na straně klienta a umožnění tvorby rychlých a dynamických aplikací a online služeb. Jde pojetí JS jako primární programovací jazyk, jehož zpracování se díky překladu do strojového jazyka několikanásobně zefektivňuje.

v8 je implementován tak, aby byl spustitelný na všech platformách(Window, MacOS, Linux) a to i v režimu standalone a může být importován do jakékoliv C++ aplikace.[8]

4.1.1 Optimalizace

v8 využívá různé optimalizační techniky, aby byl co nejrychlejší.

- **Hidden classes**
  Poskytují dodatečné informace o objektech (jejich atributech) stejného typu. Jde o simulaci tříd, protože JS třídy nezná. Každé funkci je tedy vytvořena hidden class a každý vytvořený objekt této funkce (objekt se stejnou strukturou) má stejnou hidden class. Tato třída se ukládá do paměti. Pokud přidáme nový atribut, vytvoří se nová skrytá třída z kopie té původní a na začátek se přidá onen nový atribut. K třídě se následně přidává značka, že obsahuje nový atribut a má se změnit na nově vytvořenou třídu. Tento proces je pomalý, ale provede se pouze jednou. Hidden classes umožňují jednoduše alokovat uložený atribut v paměti na místo zdlouhavého vyhledávání.[8]

- **Inline caching**
  Inline caching zrychluje vyhledávání atributu v paměti tím, že předpokládá, že atribut má stále stejnou hidden class. Při prvním přístupu zdlouhavě atribut vyhledá, ale potom jeho pozice zůstane zacachována. Pokud se skryté třídy liší, pak se vyhledává v množině skrytých tříd, ale to je stále rychlejší, než vyhledávání bez těchto informací. Při každém cyklu garbage collectoru se odmazávají kusy kódu, které vyhledávají v již nepoužívaných skrytých třídách, aby se množina co nejvíce zmenšila, pokud možno na jednu třídu.[8]

4.1.2 Garbage Collector

Tak jako Java implementuje Garbage Collector (dále jen „GC“), tak i v8 implementuje pokročilý GC. v8 využívá tzv. strict stop-the-world generační GC. Strict zde znamená, že má přesný přehled o tom, jaké objekty jsou alokované. Uchovává dvě generace objektů: mladou(menší) a starou(obsáhlejší). Každý nově alokovaný objekt je přidán do mladé generace, která je často promazávána. Pokud vytvořený objekt přežije delší dobu, pak je automaticky zařazen do starší generace, která se nekontroluje tak často. Cílem je odlišit objekty, které nemají dlouhé trvání od těch, které trvají delší dobu. Kontrolou starých generací nezískáváme moc místa v paměti, protože používané objekty nemůžeme smazat a navíc jejich kontrola je zdlouhavá. Proto se prioritně mažou objekty z mladé generace. Při mazání objektů (kontrole) se zastavuje běh programu, proto stop-the-world. [8]

GC obsahuje tři typy promazávání objektů:

- **Scavenge**
  Je to nejrychlejší collector, který prochází pouze mladou generaci objektů.
- **Full non-compacting collection**
  Středně rychlý collector, který prochází obě generace a ukládá volnou paměť do seznamů - Mark-Sweep.
- **Full compacting collection**
  Je nejpomalejší, protože dokáže fragmentovat volné místo v paměti. Prochází obě generace - Mark-Sweep-Compact.

4.2 JSON

JSON je jednoduchý formát pro předávání dat, jehož strojové zpracování je velice efektivní. Skládá se z páru key:value, kdy hodnota může být v moderních programovacích jazycích typu array, string, number, list, nebo dictionary(další key:value objekt). JSON příklad a ekvivalentní zápis v XML je v ukázce 6 a 7

```
{"users":{"martin":{"name":"Martin Lonsky","age":24}}}
```

Výpis 6: Příklad JSON serializace

```
<users>
   <martin>
      <name>Martin Lonsky</name>
      <age>24</age>
   </martin>
</users>
```

Výpis 7: XML ekvivalent k JSON

4.3 NodeJS

NodeJS je platforma, která je postavená na Chrome JavaScript runtime(v kapitole 4.1), která byla vytvořena Ryanem Dahlem v roce 2009. První publikace byla téhož roku na JSConf, kde Dahl představil NodeJS 0.1. Aplikace programované v NodeJS se píší v JavaScriptu, který je většině programátorů velice blízký a je jednoduchý k pochopení. Kód aplikace se po spuštění aplikace okamžitě celý(včetně importovaných modulů) překládá do strojového jazyka [5]. Node umožňuje vysoce škálovatelné aplikace, které jsou řízeny událostmi. Každá operace zde má na vstupu i callback, který je volán jako událost po provedení operace[6]. Platforma umožňuje neblokované Input/Outpu spojení, které umožňuje vývoj realtime aplikací s velkým objemem přenášených dat. Node aplikace pracují na TCP. Node implementuje nativně modul HTTP pro obsluhu tohoto webového protokolu a defaultně naslouchá na portu 1337[7]. HTTP zde umí pracovat s Request/Response hlavičkami a nepotřebuje k obsluze požadavků dalších softwarů, jako je Apache apod. [5]. Pro tvorbu routovaných aplikací, kde jsou již předpřipravené metody pro parsování požadavků existuje Express 4.4.

Výhodou NodeJS je, že obsahuje moduly pro práci se síťovou vrstvou a pro protokol TCP.

Další výhodou je, že NodeJS obsahuje NPM (Node Package Manager), který se stará o správu instalovaných balíků (modulů), aktualizace a správu verzí. NPM také umožňuje distribuci aplikace bez přídavných modulů, které jsou ve všech verzích volně dostupné tak, že součástí projektu je soubor `package.json`. V tomto souboru jsou ve formátu JSON zapsána data o projektu a `dependencies`. Dependencies obsahují názvy a verze modulů, které aplikace využívá. Pro distribuci pak stačí jen zdrojové kódy samotné aplikace a tento soubor. Instalace na cílové stanici probíhá spuštěním `npm install` a NPM automaticky tento soubor načte a inicializuje celý projekt včetně stažení zásuvných modulů v požadovaných verzích.

Každá aplikace v NodeJS běží jako samostatný proces/deamon a jako single process [8]. Tím se NodeJS aplikace liší třeba od PHP nad Apachem, kdy Apache naslouchá na určitém portu a požadavkům vytváří procesy. Tento proces demostruje obrázek 2

```
var fs = require('fs');
fs.unlink(<PATH_TO_FILE>, function(err){
    if (!err){
        core.EventManager().trigger('file:' + md5(<PATH_TO_FILE>) + ':removed');
    }
});
```

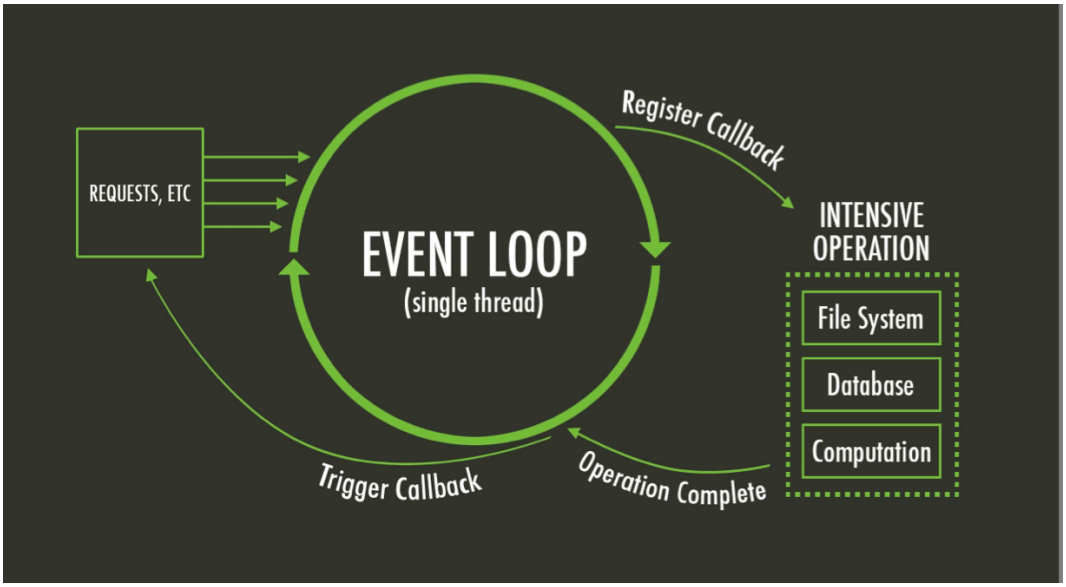
Výpis 8: NodeJS unlink event

Kód 8 ukazuje událostně řízenou aplikaci v NodeJS. Je zde využit vlastní EventManager, který je blíže popsaný v kapitole 7. NodeJS je dnes velice rozšířené a používá jej například Microsoft ve svém Azure, nebo LinkedIn mobile.

[5] NodeJS aplikaci lze zredukovat i do jednoho spustitelného souboru pomocí nexe [7].

[6] Některé moduly umožňují také synchronní operace, jako je například existsSync u modulu File System

[7] Na Portu 80 typicky naslouchají jiné deamony: Apache atd.



Obrázek 2: NodeJS single process [12] - demo 8

```
var http = require('http');
http.createServer(function(req, res){
    res.writeHead(200, { 'Content-Type' : 'text/plain' });
    setTimeout(function(){
        res.end('works!');
    }, 5000);
    res.write('It ');
}).listen(3000);
```

Výpis 9: NodeJS server a non-blocking I/O

V ukázce 9 jsem vypíchl vlastnost non-blocking I/O v NodeJS. Klientovi se zobrazí na obrazovce v plain textu „It “ a teprve o pět sekund později se spojení uzavře se zbytkem zprávy „works!“.

Spouštění aplikací v NodeJS se dále věnuje kapitola 4.6.

4.3.1 Alternativy k NodeJS

Alternativou k NodeJS může být například Vert.x. Zatím co NodeJS je vytvořen použitím JS, Vert.x je postaven na Javě. Vert.x je do jisté míry inspirováno NodeJS, ale má svou vlastní filosofii a nemůžeme říci, že Vert.x je NodeJS napsané v Javě.

Stejně jako NodeJS běží jako serverový proces(deamon). Další společnou vlastností je událostně řízený proces a tedy asynchronní API. Na rozdíl od NodeJS, pro který se apliakce píši v JS, protože implementuje Google v8(4.1), Vert.x aplikace je možné programovat v různých programovacích jazycích jako je `Python`, `Ruby`, ale také `JS`.

Podstatnou změnou oproti NodeJS je multi-threading. NodeJS pracuje jako jediný proces, ačkoliv v komunitách vznikají možnosti vícevláknových procesů a v dohledné době to bude podporovat nativně samotné NodeJS. Vert.x dokáže pracovat ve více vláknech k využití všech jader procesoru bez nutnosti synchronizace a blokování.

Jednotlivé aplikace(deamoni) mezi sebou mohou jednoduše komunikovat a nezáleží na tom, v jakém jazyce jsou implementovány. To je zajištěno pomocí open-source In-Memory Data Grid.

Jednotlivé moduly mohou být naskrz různými aplikacemi sdíleny a taktéž nezáleží na programovacím jazyku[6].

4.4 Express framework

Express je web application framework vyvinutý pro platformu NodeJS. Je to jednoduchý framework určený k vývoji internetových aplikací na této platformě. Obsahuje škálu implementovaných submodulů pro správu obsahu(například: CookieParser, SessionStorage). Express umožňuje také renderování šablon pomocí různých templatovacích enginů. Nejznámějším templatovacícm enginem na této platformě je Jade 10.

```
doctype html
html(lang="cs")
    head
        title = title
    body
        h1 Jade - Template engine
        #nejakyDiv.nejakaTrida
            if podminka
                p splnena
            else
                p nesplnena
            p Odstavec
```

Výpis 10: Template v Jade

Tento způsob templatování je velice praktický a jde spíše o programátorský přístup. Alternativou JADE je EJS, který také využívá můj projekt. Jde o jednodušší variantu, která zachovává klasickou strukturu HTML kódu. Předávané proměnné se v templatech vypisují způsobem připomínajícím ASP.NET a to `<%- promenna%>`. Samozřejmě i EJS nabízí možnosti podmínek, cyklů, importu CSS a JS souborů a další funkce. EJS jsem zvolil proto, abych dříve, či později nenarazil na něco, co se pomocí JADE dá jen těžko realizovat [9]. Výhodou JADE je to, že podle deklarace typu dokumentu(doctype[10]) bude vygenerovaný template validní a bez zbytečných prvků(nodes) navíc. [10] [11]

Ve své podstatě je Express pouze Request/Response handler s rozšířenými možnostmi. U routování lze nastavit formát URI a pokud URI požadavku souhlasí, pak je volán callback `app.get('/editor', function callback)`. Routování je rozděleno na GET a POST požadavky a mezi sebou se nemísí. POST požadavek není obsloužen routou pro GET, ačkoliv URI souhlasí.

V ukázce 11 je tvorba aplikace a route handleru s Express frameowrkem.

```
var express = require('express'),
    http = require('http'),
    app = express();
app.get('/', function(req, res){
    res.end('It works!');
    /*res.render('NazevTemplatu', { var1 : 'Promenna1 do templatu'}); - pokud mame
        zaregistrovany nejaky template engine (JADE, EJS, HTML, ...)*/
});
var server = http.createServer(app).listen(3000, function(){
```

[9] Například nějaký úmyslný HTML/CSS hack pro docílení nestandardního/tvůrci nezamýšleného chování.

[10] Volba standardu XML dokumentu.

```
    console.log('Listening on port: 3000');
});
```

Výpis 11: Vytvoření Express aplikace a routing

4.5 Socket.IO

V kapitole 2.2.1 byla zmíněna funkce `WebSocket` pro práci s WS. Některé prohlížeče však tuto funkci nativně neznají, nebo se jmenuje jinak(např: FireFox má `MozWebSocket`). Socket.IO je API vytvořené jak pro server, tak pro klienta, které zastřešuje WS. Umožňuje tak komunikaci přes WS skrze veškeré typy zařízení a odstiňuje tak vývojáře od řešení odlišností mezi prohlížeči. Taktéž nezáleží na typu transportu dat[19].

Velikou výhodou je, že Socket.IO API je implementováno jak v JS pro klienta, tak pro NodeJS. Sjednocuje tak implementaci a názvosloví funkcí a událostí mezi serverem a klientem. To značně usnadňuje orientaci ve zdrojovém kódu viz. implementace serveru12 a klienta12.

```
var io = require('socket.io').listen(server);
io.on('connection', function(socket){
    //probehl handshake
    io.on('message', function(data){
        io.broadcast.emit('message', data);
    });
});
```

Výpis 12: Socket.IO server implementace

Na straně serveru je několik důležitých možností ke konfiguraci. Jednak již zmíněné nastavení typů transportů(`websocket`, `flashsocket`, `htmlfile`, `xhr-polling`, `jsonp-polling`, ...), také nastavení autorizace a intervalu aktualizace handshake pro udržení spojení(tzv. heartbeat) a v neposlední řadě nastavení úložiště pro vytvářené spojení. V editoru využívám RedisServer, který je dále v kapitole 4.5.1.

```
var io = new window.io.connect('http://example.com:3000');
io.on('connect', function() {
    //probehl handshake
    io.on('message', function(data){
        console.log('New message: ' + data);
    });
    io.emit('message', 'It works!');
});
```

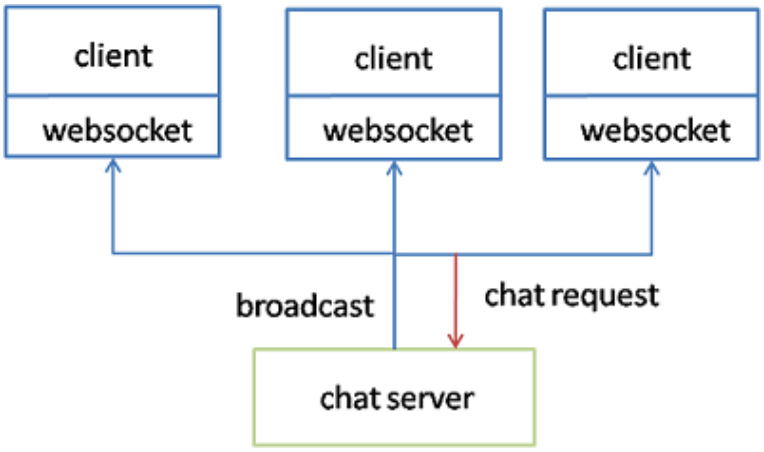
Výpis 13: Socket.IO client implementace

Socket.IO umožnuje měnit typy transportu mezi klientem a serverem. Lze striktně nařídit serveru, že má používat jeden konkrétní typ transportu nebo množinu zvolených. Pokud zvolíme širší množinu povolených typů transportu, docílíme tak vyšší kompatibility mezi typy prohlížečů. Pokud totiž prohlížeč nepodporuje WebSocket, pak je automaticky nahrazen AJAXem3.

Stálá kompatibilita mezi implementací Socket.IO u klienta a implementací na serveru je zajištěna tím, že aktualizacím podléhá pouze modul Socket.IO na serveru. Socket.IO se totiž automaticky přidává do routování na straně serveru a je klientovi generován aktuální JS kód tak, jak je to demostrováno zde 14

```
<script src="/socket.io/socket.io.js"></script>
```

Výpis 14: Socket.IO import u klienta



Obrázek 3: Socket.IO Server/Clients [20]

Na obrázku 3 je zobrazena komunikace mezi klienty a serverem. Jelikož NodeJS pracuje jako single process, lze využít broadcastingu. Broadcasting zde znamená, že server odesílá zprávu všem stanicím kromě té, ze které byl broadcasting vyvolán.

4.5.1 RedisServer

RedisServer(dále jen „Redis“) je open-source projekt, který umožňuje pokročilé ukládání klíč:hodnota(key:value) párů. Může obsahovat datové struktury serveru, protože hodnota může nabývat nejen řetězce a čísla, ale také seznamy(lists), ale také sety a tříděné sety. Redis Client API je implementováno pro většinu známých programovacích jazyků, jako je `C`, `Java`, `PHP`, `Python`, ale také `NodeJS`(v kapitole 4.3).

Redis je více než jen Cache server, jak si mnoho lidí myslí. Dokáže výsledky různě třídit a vypisovat jen potřebná data. Neobsahuje sice pokročilé možnosti jako SQL, nebo nonSQL databáze, ale zato je mnohonásobně rychlejší, a proto se používá ke Cachování.

Rozdíly, například oproti Memcached:

- Cache zůstane uložená i po restartu serveru.
- Více možností, jaké typy dat můžeme ukládat.
- Selektivní mazání položek.

V editoru RedisServer používám k ukládání a cachování socket poolu.

4.6 Spouštění aplikací

V kapitole 4.1 bylo řečeno, že Google v8 je implementován pro všechny známé platformy. Taktéž NodeJS, který je postaven na v8 je implementován pro všechny platformy. Spouštění probíhá na všech platformách stejně. Je zapotřebí mít správně nastavenou cestu k NodeJS kompilátoru v systémových proměnných(enviroment variables). Ukázka spuštění na OS Linux 15.

```
# nodejs <APP_PATH>.js [arg2, ...]
```

Výpis 15: Bash - Spuštění aplikace

Vstupní parametry jsou číslovány od 1, kdy arg1 je název spouštěného souboru. V případě implementovaného editoru je další nepovinný [11] argument port, na kterém aplikace naslouchá.

Aplikace se spouštějí nad konkrétním souborem, který si volá další moduly a vytváří server naslouchající nad zvoleným portem. Z tohoto důvodu je nutné pro zachování procesu využít dalších utilit pro běh procesu na pozadí. Na Linuxu pro to slouží screen.

Jelikož je NodeJS aplikace kompilovaná do strojového jazyka, všechny syntaktické chyby, nebo chybějící moduly se projeví již při samotném spuštění. Při kompilaci se překládají pouze knihovny a třídy do strojového jazyka, nikoliv samotné instance, takže přirozeně všechny ostatní runtime chyby je nutné ladit až v průběhu chodu samotné aplikace.

[11] Pokud není definováno, pak výchozím portem je port 3000

5 MongoDB

MongoDB (dále jen „Mongo“) je open-source nonSQL[12] databáze implementována v C++. nonSQL databáze jsou novým trendem v oblasti vývoje informačních systémů a pojí se typicky s databázemi, které nemají fixní schéma. Tyto databáze mají typicky nižší bezpečnost transakcí, ale selekce dat z nich je daleko rychlejší než u SQL databází. Mongo byla vytvořena v roce 2009 jako jedna z prvních nonSQL databází. Díky rychlosti čtení a zápisu dat v nonSQL databází jsou tyto databáze vhodné pro real-time aplikace s velkou intenzitou čtení a zápisů do databáze.

Jelikož schéma databáze není fixní, ukládají se pouze ta data, která jsou databázi předána. Neexistuje, jako u SQL databází, že některá nepovinná pole zůstanou s hodnotou `NULL`. Tato pole nemusí být vůbec do databáze ukládána a programově pak neexistenci pole patřičně vyhodnotíme.

Mongo ukládané položce automaticky tvoří unikátní indexy `_id`, které jsou tvořeny jako `ObjectId(rand)`. Tytoto indexy jsou implementovány jako B-Tree.[13]

Příkazy v Mongo se formulují pomocí JS syntaxe a veškeré vstupní parametry jsou buď String, Number nebo JSON viz. 4. Na obrázku je patrné, že práce s databází funguje principiálně stejně, jako u SQL databází. Jednotlivé „tabulky“ jsou zde zastoupeny kolekcemi(collections).

Mongo je samozřejmě také typově orintované. `"1"` a `1` rozhodně nejsou porovnatelné operátory ===, ani ==[13]

```
root@cobra4:~# mongo
MongoDB shell version: 2.0.6
connecting to: test
> db.test.insert({name: 'Martin Lonsky', login: 'visitek'})
> db.test.find()
{ "_id" : ObjectId("535f8cbf089196d41784f74a"), "name" : "Martin Lonsky", "login" : "visitek" }
> db.test.update({'_id': ObjectId("535f8cbf089196d41784f74a")}, {$set: {name: "Lonsky Martin"}})
> db.test.find({'_id': ObjectId("535f8cbf089196d41784f74a")}, {name: 1})
{ "_id" : ObjectId("535f8cbf089196d41784f74a"), "name" : "Lonsky Martin" }
> db.test.remove({'_id': ObjectId("535f8cbf089196d41784f74a")})
> db.test.find()
> db.test.drop()
true
>
bye
root@cobra4:~#
```

Obrázek 4: MongoDB console

V JS je možné nechat si vypsat tělo funkce tak, že vypíšeme referenci na funkci, kterou nevoláme. Téhož lze docílit v Mongo konzoli 5, kde je v příkazu nevolaná funkce, která se vypíše do logu. Tělo vypsané funkce, která se stará o přepis dat v databázi je dle obrázku implementováno v JS.

Data jsou fyzicky uložená pomocí BSON. BSON je binary-encoded serializace JSON dokumnentu. BSON obsahuje rozšíření, které umožňuje ukládání stejných datových typů,

[12] K manipulaci s databází se nevyužívá SQL jazyka.

[13] Na rozdíl od automatických konverzí v některých programovacích jazycích, jako je PHP.

```
> db.test.update
function (query, obj, upsert, multi) {
    assert(query, "need a query");
    assert(obj, "need an object");
    var firstKey = null;
    for (var k in obj) {
        firstKey = k;
        break;
    }
    if (firstKey != null && firstKey[0] == "$") {
        this._validateObject(obj);
    } else {
        this._validateForStorage(obj);
    }
    this._mongo.update(this._fullName, query, obj, upsert ? true : false, multi ? true : false);
}
```

Obrázek 5: MongoDB konzole - vypsání kódu

jako JSON. Konverze do/z BSON probíhá velice rychle ve většině programovacích jazyků díky využití datových typů z C[15].

Mongo je v současné době stále v prudkém vývoji. V poslední verzi 2.6 byla přidána možnost omezování práv uživatelům na konkrétní kolekce.

5.1 NodeJS

Pro NodeJS existuje asynchronní adapter. Práce s kolekcemi je zde velice jednoduchá. Nad objektem vytvořeného spojení k databázi stačí zavolat funkci `collection` s parametrem názvu kolekce, což nahrazuje zápis `db.<collection>` z obrázku 4. Následně lze přistupovat k datům pomocí stejných funkcí s tím rozdílem, že musíme v parametru vždy předávat i callback(protože jde o asynchronní adapter)[14].

Ukázka výpisu data podle `_id` v NodeJS16 se trochu liší od výpisu dat podle `_id` v konzoli5

```
db.collection('test').find({'_id': new db.ObjectID('535f8cbf089196d41784f74a')})
    .toArray(function(data){
        console.log(data);
    });
```

Výpis 16: MongoDB v NodeJS

[14] Existuje module mongo-sync, který skrze Fibers přidává možnost používat MongoDB synchronně.

6 Google diff_match_patch

diff_match_patch je API vytvářena Googlem pro porovnávání plain-textu a zjištění rozdílů. Tato API je momentálně dostupná pro `Java`, **JavaScript**, `Dart`, `C++`, `C#`, `Objective C`, `Lua` a `Python`.

Knihovna se skládá ze 3 částí:

- **Diff**
  Porovná dva bloky plain-textu a efektivně vrátí seznam rozdílů.
- **Match**
  Vyhledá podle zadaného řetězce nejlepší shodu(tzv. fuzzy match) v bloku plain-textu. Tato shoda je vážena jak pro přesnost, tak pro umístění v textu.
- **Patch**
  Aplikuje vygenerovaný seznam rozdílů na porovnávaný text tak, aby oba porovnávané bloky textu byly shodné.

Toto API implementuje Meyersův rozdílový algoritmus dále popsaný v kapitole 6.1.

6.1 Meyersův rozdílový algoritmus

Eugene W. Meyers(1953) je americký počítačový expert a bioinformatik.

Problém transformace jednoho textu tak, aby z něj vznikl text shodný s druhým, je dlouhodobě známý problém. Je dokázáno, že algoritmus má časovou složitost $O(N+D^2)$, kde N je délka, A+B a D je velikost nejmenší úpravy pro A a B.

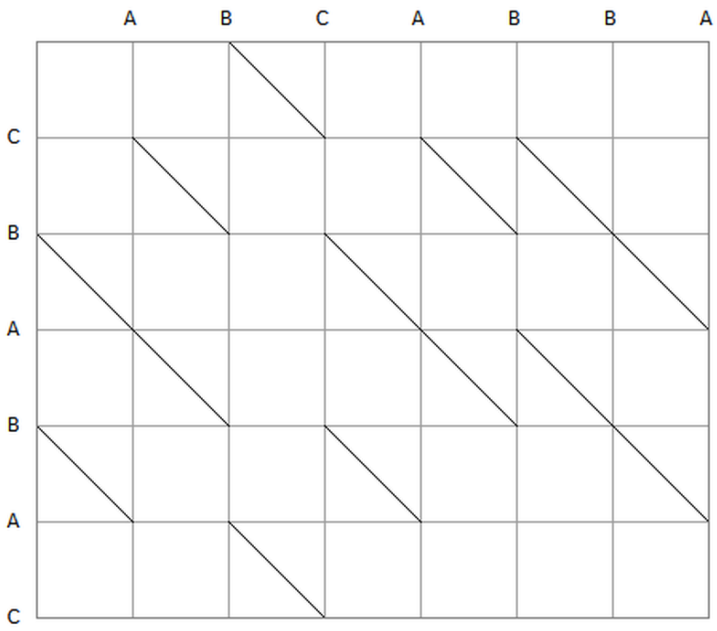
Meyersův algoritmus hledá nejkratší sekvenci úprav tak, aby pomocí operací delete a insert převedl text A na text B. Pro následující příklad využiji stejné řetězce, jako Meyers využil ve své publikaci z roku 1986[4].

```
text A = ABCABBA
text B = CBABAC
```

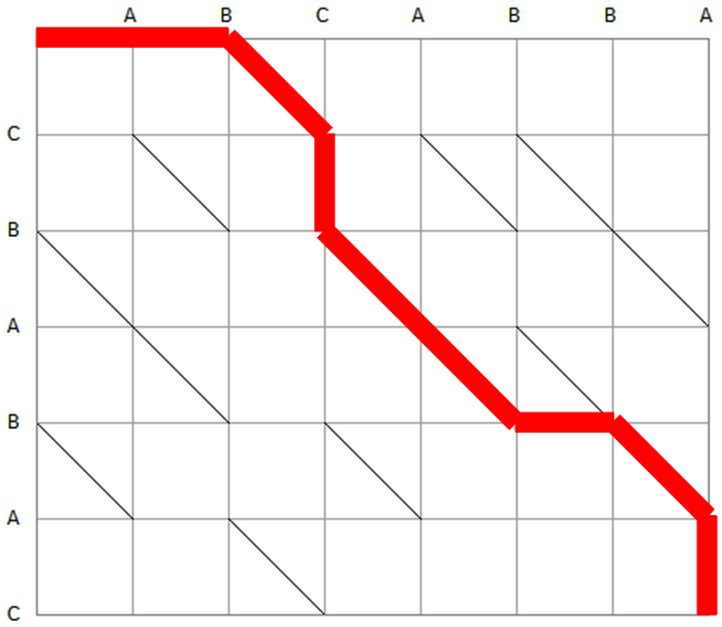
Předchozí dva texty převedeme do pole po znacích: `A[]` o velikosti `N` a `B[]` o velikosti `M`. Jednotlivé hodnoty polí zaneseme do grafu tak, že A naneseme na osu x a B na osu y, kdy indexy číslujeme od 1 do `N` a `M`. V grafu nám posun doprava po x bude znázorňovat operaci delete a posun dolů po y operaci insert. Algoritmus tedy hledá cestu z levého horního rohu `(0,0)` do pravého dolního rohu `(N,M)`. Následně zaznačíme do grafu všechny jednokrokové diagonály spojující jednotlivé přechody mezi následujícími znaky tak, jak je to na obrázku 6.

Dalším postupem je nalezení nejkratší cesty z bodu `(0,0)` do `(N,M)` tak, že je možné se vydat dvěma směry. Jak po ose x, tak po ose y. Posun může probíhat pouze horizontálně a vertikálně, nebo, je-li v grafu zaznačena diagonální cesta, tak diagonálně.

Na obrázku 7 je znázorněna nejkratší cesta reprezentující operace:



Obrázek 6: Meyersův graf nalezení nejkratší cesty



Obrázek 7: Meyersův graf nejkratší cesta

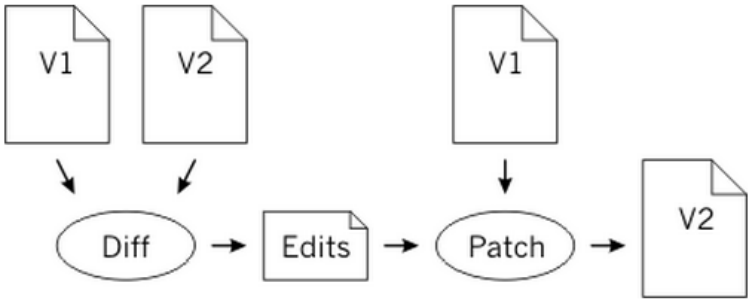
1. delete(0)
2. delete(0)
3. ~~delete(0), insert(C, 0)~~
4. insert(B, 1)
5. ~~delete(2), insert(A, 2)~~
6. ~~delete(3), insert(B, 3)~~
7. delete(4)
8. ~~delete(4), insert(A, 4)~~
9. insert(C, 5)

První parametr operace je pozice, se kterou se manipuluje. Operace insert obsahuje navíc druhý parametr, kterým je přidávaný znak na danou pozici.

Můžeme si všimnout, že diagonální posun je možné vyškrtnout, protože dochází ke smazání a následně k přidání stejného znaku. Nejkratší sekvence úprav má tedy 9 operací a z toho 4 přeskoky - tedy 5 operací, které se provedou.

6.2 Fuzzy patch

Podívejme se na obrázek 8.



Obrázek 8: Fuzzy patch

Patch je velice jednoduchý proces, který aplikuje změny v dokumentu V1 na dokument V2 podle vygenerované sekvence změn, popsané v kapitole 6.1, tak, aby výsledný dokument V2 byl shodný s dokumentem V1. Za předpokladu, že dokumenty V1 jsou totožné, je proces velice jednoduchý. Pokud se ale dokumenty V1 liší, pak se situace značně komplikuje, protože algoritmus musí jak nejlépe umí najít pozici, na kterou patch aplikuje v dokumentu V2. V podstatě se dokument V1 může po vygenerování diff ještě změnit a při aplikování patch se již liší od původního.

Standardem pro aplikovaní patche je GNU patch. GNU patch uchovává na začátku sekvence změn informaci o tom, na jaké řádky má být změna aplikována. Pokud řádky nesouhlasí(právě z důvodu změny V1), pokusí se patch nalézt novou pozici. Pokud pozice není nalezena, patch se pokusí vyhledat pozici bez prvního a posledního řádku patche. Pokud opět není nová pozice nalezena, pokračuje bez prvních a posledních dvou atd. Tato konstrukce je docela křehká, protože i malé změny mezi dvěma dokumenty mohou zapříčinit selhání i když ostatní řádky sedí.

Sofistikovanější přístup, který by zvýšil počet úspěšných shod a snížil počet selhání, by byl znakově založený přístup místo řádkový. V klasickém textu je 2% pravděpodobnost, že se stejný kus kódu objeví ve stejné kapitole. Pokud ovšem pracujeme s nějakým zdrojovým kódem, pak se můžeme setkat s vícenásobným opakováním stejného řádku. Není tedy možné určit nejoptimálnější délku sekvence znaků k jednoznačné identifikaci správné pozice. Proto algoritmus přidává po čtyřech znacích z každé strany dokud blok nebude v textu unikátní. Minimální délka je tedy 8 znaků a maximální je délka dokumentu[23].

6.2.1 Levenshteinova vzdálenost

Neboli také editační vzdálenost. Vladimír Levenshtein ji zavedl v roce 1965. Je definovaná jako minimální počet operací delete, insert nebo nahrazení tak, aby po jejich provedení byly výsledné dva řetězce shodné. Podobně jako v Hammingově vzdálenosti[15][24].

*a*m*ul*et a *u*m*lc*et - počet nutných operací je **3**

Distanční tabulka obsahuje tolik sloupců, kolik je délka celého řetězce, ve kterém je vyhledáván podřetězec. Jako řádky tabulky je délka vyhledávaného podřetězce. V tabulce probíhá posun následovně[16][25]:

- **Shodný průsečík** - operace **identical** = posun diagonálně
  `value(i, j) = value(i-1, j-1)`
- **Odstranění znaku** z prohledávaného textu - operace **delete** = posun horizontálně
  `value(i, j) = value(i, j-1)+1`
- **Vložení znaku** do prohledávaného textu - operace **insert** = posun vertikálně
  `value(i, j) = value(i-1, j)+1`
- **Substituce znaku** prohledávaného textu - operace **substitute** = posun diagonálně
  `value(i, j) = value(i-1, j-1)+1`

[15] Počet pozic, které neobsahují shodný znak ve dvou řetězcích stejné délky - počet nutných operací k dosažení dvou totožných řetězců stejné délky.

[16] Prohledávaný text se průběžně modifikuje.

7 jQuery - EventManager plugin

U realtimových aplikací je nutné reagovat v jednotlivých modulech na události, které se provedou v jiných modulech. Jak již bylo zmíněno v kapitole 4.3, NodeJS pracuje asynchronně a po provedení operace volá callback. EventManager se stará o reakce na dokončení operací tak, že po provedení události je vyvolána událost(event) s určitým id(může být String) události. Na události může být navázán neomezený počet posluchačů(lsitenerů), kteří čekají, až bude událost vyvolána a poté provedou svou část kódu, která je k posluchači předána tak, jak to demonstruje kód 17.

```
$.EventManager.attach('event1', function(txt  [,  arg1 [,  arg2 ...]]) {
    console.log('Triggered: ' + txt);
}, true);
//Na jinem miste v  aplikaci  nasledovne
$.EventManager.trigger('event1',  'Event1' [,  arg1 [,  arg2 ...]]) ;
```

Výpis 17: Demonstrace využití EventManageru

Příkladem využití EventManageru je smazání souboru, které je implemetováno ve třídě `File`. NodeJS module `FileSystem` se pokusí soubor smazat a pokud se mu to podaří, pak je vyvolána událost smazání určitého souboru. Každé vytvoření spojení v Socket.IO má vytvořené lsitenery pro manipulaci s konkrétním souborem a tedy naslouchá i jeho smazání. Teprve po obdržení signálu, že se smazání zdařilo, jsou ostatní stanice pomocí broadcastingu spraveny o smazání souboru.

Výhodou tohoto EventManageru je, že listenery je možné shlukovat do pracovních skupin(workspaces) pomocí tagování. Lze tedy vyvolat událost jak podle eventid, tak i podle workspace.

Nejvíce je však využívaná možnost smazání všech listenerů podle určité workspace. Příkladem workspace může být `hash` názvu souboru. Tato workspace obsahuje listenery, kteří naslouchají událostem vyvolaným manipulací se souborem(rename, create, unlink). Pokud je soubor uzavřen nebo smazán, je nutné tyto listenery z paměti vyčistit a tedy zavoláme `detahByTag('file:' + hash)`.

Funkce `detahByTag` je také využívána pro odstranění jednorázových listenerů, které za určitých nesplněných podmínek nebyly volány a s dalším přidáním jednorázového listeneru by se listenery vrstvily, což je nežádoucí. Jednorázový listener se provede jen jednou a poté je smazán a již nenaslouchá. Pokud se ale neprovede, pak smazán není a čeká. Workspace může obsahovat i nejednorázové události, proto je vhodnější využít spíše `detahAllOnceByTag`

EventManager je implementován pro jQuery framework, ale je využíván také jako modul v implementovaném editoru [17].

jQuery verze je dostupná na mém githubu.

[17] Deklarace pro $ je nahrazena za module.export pro NodeJS modul.

8 rEditor.io - realtime editor

rEditor.io je pracovní název pro vzniklý real-time editor postavený na technologii WebSocket(2.2) zastřešenou API Socket.IO(4.5). Je to nástroj ke správě serverově lokálního souborového systému. Hlavní motivací pro tvorbu takového nástroje byla myšlenka sjednocení více nástrojů dohromady a rozšířit je o možnost kontinuální práce více uživatelů zároveň. Díky pokročilých možnostem a technikám, které nabízí HTML5(2), je možné simulovat chování desktopových aplikací na webové stránce.

8.1 Hlavní cíle projektu

- Nahradit využívání nástrojů pro FTP při tvorbě a editaci webových stránek.
- Usnadnit orientaci ve zdrojovém kódu.
- Usnadnit zaškolení do stávajícího API pro tvorbu webových stránek. Může sloužit jako prezentační software k demonstraci tvorby webu na dálku.
- Umožnit jednoduchou kontrolu práce díky historii s časovými razítky.

Výsledná aplikace musí splňovat následující předem stanovené požadavky:

- **Dostupnost**
  Aplikace musí být dostupná odkudkoliv bez nutnosti lokální instalace.
- **Bezpečnost**
  Aplikace musí obsahovat autentifikační a autorizační metody pro zajištění bezpečnosti lokálního prostoru.
- **Adaptivní a personalizovaný obsah**
  Jednotlivým uživatelů je poskytnut diferencovaný obsah.
- **Bezpečný transport**
  Metody pro transport dat a souborů na server a ukládání na serveru.
- **Historie**
  Uchování historie manipulací se soubory na serveru a možnost porovnání verzí.
- **Komunikace**
  Možnost komunikace uživatelů mezi sebou.
- **Rozpoznání typů souborů**
  Rozpoznávat typy souborů a umožnit editaci obsahu pouze textových.

- **Simulovat desktopové programy**
  Zvýraznění syntaxe zdrojových kódů.

8.2 Výhody - Strengths

- Aplikaci není potřeba instalovat a je dostupná odkudkoliv.
- Aplikace shlukuje více programů do jednoho.
- Server uchovává data, aktuálně otevřené soubory a přihlášené uživatele. Pád prohlížeče nebo celého hardwaru na straně klienta neovlivní nedokončenou práci.
- Kompatibilita naskrz všemi platformami.
- V případě aktualizace se změny projeví okamžitě.

8.3 Nevýhody - Weaknesses

- Nutnost stabilního připojení k internetu[18].
- Rozdílné chování jednotlivých internetových prohlížečů.

8.4 Požadavky a jejich specifikace

Analýza požadavků je důležitou částí životního cyklu implementace informačních systémů(dále jen „IS“). Za základně analyzovaných požadavků a výsledné sady funkcí, které musí výsledný IS obsahovat, volíme vhodné technologie pro implementaci.

8.4.1 Funkční požadavky

- Aplikace musí umět spravovat souborový systém(`FileSystem`) jako FTP nástroje.
- Aplikace musí umět editovat soubory s rozšířenou možností zvýrazňování syntaxe známých souborů.
- Aplikace musí umět pracovat v online režimu a reagovat tak na změny na ostatních stanicích.
- Aplikace musí umožňovat komunikaci mezi pracovníky.
- Aplikace musí umět personalizovat data a chránit tak data v hyperprostoru před neoprávněnými přístupy.

[18]Offline práce je předmětem další diskuse.

8.4.2 Sada základních funkcí

Sada základní funkcí vychází z analýzy desktopových aplikací. Výsledná aplikace musí nabízet základní funkce, které nabízejí desktopové editory. Hlavním inspirátorem byl software PhpStorm společnosti JetBrains s.r.o., jehož licenci mám zakoupenou, určený primárně pro vývoj PHP aplikací. Nabízí také spoustu zásuvných modulů, mezi které patří i NodeJS. Screenshot PhPStorm je na obrázku 20 a 21 v příloze.

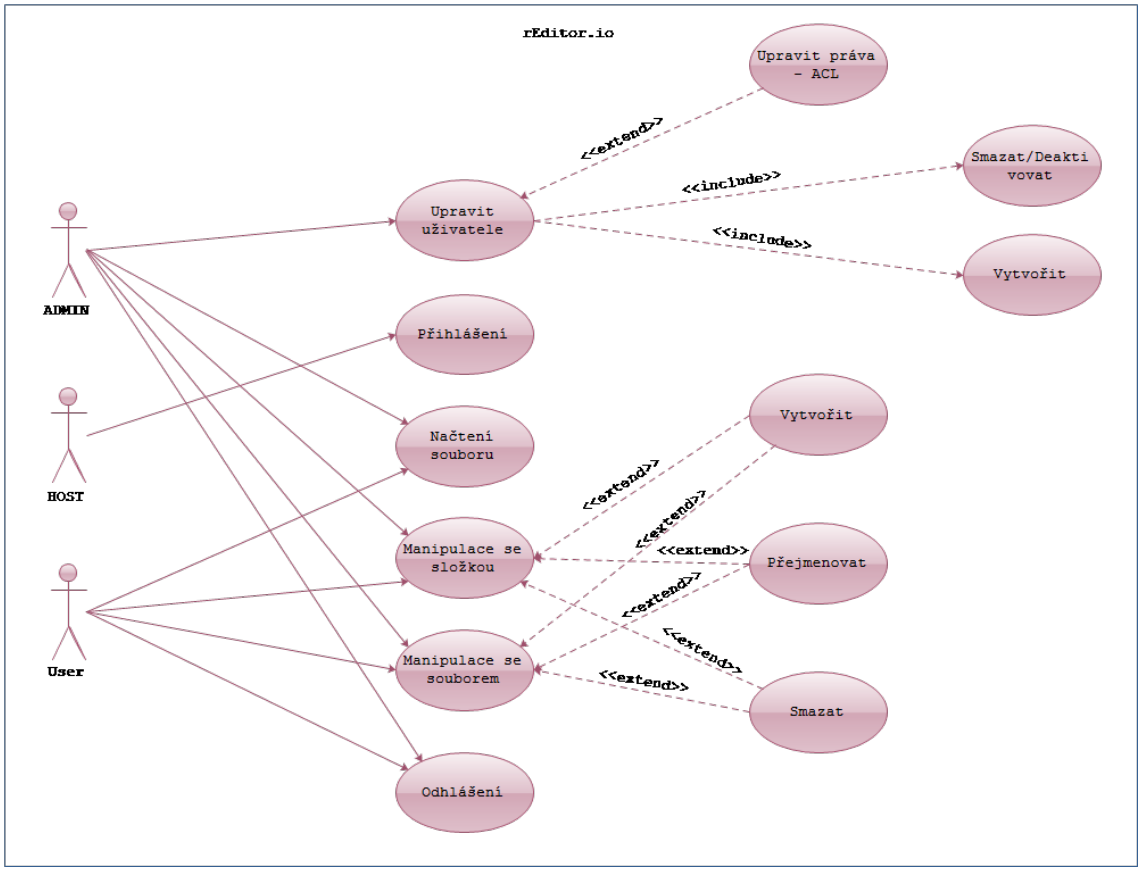
- Host, který není přihlášen je k přihlášení vyzván. Aplikace musí zamezit jakémukoliv přístupu bez validní autentifikace8.9.
- Host se autentifikuje pomocí unikátního loginu a hesla, přičemž je zapotřebí uživatelské účty chránit a případně blokovat.
- Přihlášený uživatel(dále jen „uživatel“) má možnost přidávat, přejmenovávat a mazat složky(podléhá ACL8.10).
- Uživatel může vytvářet, mazat a přejmenovávat soubory(podléhá ACL8.10).
- Uživatel může uploadovat nové soubory(podléhá ACL8.10).
- Uživatel může otevírat soubory známých typů a upravovat jejich obsah(podléhá ACL8.10).
- Uživatel může zavírat otevřené soubory skrze záložky.
- Uživatel může uložit soubor a vytvořit tak záchytný bod v historii.
- Uživatel se může vracet k předešlým verzím souboru nebo s nimi porovnávat.
- Uživatel může využít funkce back a forward i na změny jiných uživatelů.
- Uživatel může posílat zprávy v chatu ostatním uživatelům.
- Uživatel se může z aplikace odhlásit.

8.4.3 Nefunkční požadavky

Výsledná aplikace musí být dostupná odkudkoliv a musí být přístupná z různých operačních systémů. Komunikace mezi klientem a serverem musí být velice rychlá a přenos dat by měl být minimální. Komunikace mezi serverem a klientem je v kapitole8.6.

8.5 Use-case

Use-case diagram je na obrázku 9 a definuje aktéry, kteří s aplikací budou pracovat.



Obrázek 9: Use-case diagram

8.6 Real-time

Jedním z hlavních požadavků na výslednou aplikaci je možnost komunikace se serverem v reálném čase. Jakýkoliv požadavek na změnu ve `FileSystem`u(dále jen „FS“) musí být okamžitě proveden a distribuován ostatním stanicím, které se změnou souvisejí. Jestli stanice se změnou souvisí je určeno pomocí ACL8.10.

8.6.1 Volba vhodné technologie

Na základě funkčních a nefunkčních požadavků a také seznamu základních funkcí, které musí výsledná aplikace splňovat, volíme vhodné technologie. Z požadavků je patrné, že pro docílení multiplatformního použití a dostupnosti odkudkoliv je jedinou možností aplikaci stavět jako webovou aplikaci. Samotná filosofie navrhované aplikace je postavena na stálém připojení k internetu. Při volbě vhodné technologie se tedy musí brát v úvahu, že aplikace pracuje v těsném kontaktu se serverem a musí reagovat na jakoukoliv změnu na serveru. Musíme tedy zvolit technologii, která udržuje stálé spojení. Jde nám tedy

o okamžitou reakci na změny distribuované serverem, nikoliv jen o distribuci změn na požádání klienta.

Z tohoto důvodu je potřeba zvolit některou z non-blocking technologií, která umožňuje práci s `WebSocket`(2.2). V dnešní době rapidně se rozvíjející technologií je NodeJS(4.3), která všechny dříve zmíněné požadavky splňuje a díky svému jádru v Google v8(4.1) nabízí vysoký výpočetní výkon s možnosti obsluhy velkého množství koncových stanic ve stejném čase.

Pro práci s WS volím Socket.IO, což je zásuvný modul k NodeJS, určený právě ke komunikaci Client/Server pomocí WS.

8.6.2 Spojení se serverem

Aplikace je navržena tak, že vytváří nejméně jedno WS spojení, které setrvává v aktivitě po celou dobu běhu aplikace v prohlížeči. Bez tohoto spojení nelze s aplikací pracovat. Toto spojení slouží pro:

- **Řízení otevřených položek**
  Je běžné, že internetové prohlížeče z nějakého důvodu „spadnou“. Z důvodu nějaké chyby při zpracovávání některé webové stránky nebo jsme nuceni práce na okamžik zanechat z jiného důvodu. Po pádu nebo vypnutí prohlížeče nás sice aplikace odhlásí, ale po dobu ochranné doby(30 minut) naše neuložené změny nejsou ztraceny. Jakmile se přihlásíme do aplikace, okamžitě jsou automaticky otevřeny veškeré záložky, které jsme měli naposledy otevřeny, a aktuální stav neuloženého souboru. Princip pročišťování je dále popsán v kapitole 8.11.

- **Synchronizace s FS**
  Jakmile některý z uživatelů odešle jeden z požadavků na operaci: `mkdir`, `rmdir`, `touch`, `unlink`, `rename`, jsou tyto změny poslány do aplikace právě tímto spojením. Každá změna se v aplikaci projevuje až tehdy, kdy je zpracována serverem a do stanice, ze které byl požadavek odeslán, přichází ve stejné podobě, jako ostatním stanicím.

- **Upload souborů**
  Toto spojení je také zodpovědné za upload souborů popsaný v kapitole 8.8.

Každý vytvořený socket čeká, jakmile bude vyvolána nějaká událost, tedy něco se změní ve FS. Jakmile dojde k nějaké změně, postupně tato událost projde všemi listenery(u všech socketů = připojených uživatelů) a aplikace se přitom táže ACL, jestli tato změna s uživatelem souvisí. Pokud s ním souvisí, jsou data o změně zaslána uživateli socketem.

8.6.3 „File Socket“

Kromě řídícího socketu, který je otevřený po celou dobu běhu aplikace se v průběhu práce otevírají i další sockety. Aplikace vnímá každý otevřený soubor jako samostatný socket. Komunikace jednotlivých upravovaných souborů je od sebe oddělena, což zajišťuje samostatný tunel WS pro komunikaci pro každý soubor.

Pokud se nad tím zamyslíme více, může se to zdát jako zbytečnost. Synchronizaci by přece zvládl i hlavní řídící socket, kdybychom kromě seznamu změnových operací(diff patch) zaslali informaci identifikující upravovaný soubor. Ušetřili bychom tím na počtu spojení a na velikosti přenášených dat, protože Socket.IO si v definovaném intervalu stále udržuje handshake pomocí heartbeat(viz. 4.5). Podívejme se na to tedy z druhé strany. Každé vytvořené spojení se zároveň přihlašuje(18) k workspace s názvem `md5(<FILE_PATH>)` (dále jen „hash“). Pokud bychom celou komunikaci chápali jako chat, potom to znamená, že se vytvořené spojení přidalo do chatovací místnosti vytvořené pro konkrétní soubor. Mezi jednotlivými stanicemi, které mají otevřený konkrétní soubor, se tedy velice jednoduše distribuují změny. Stačí jednoduše vytvořit broadcast do celé workspace a data se odešlou na ty stanice, které naslouchají v dané „místnosti“. Můžeme mít souborů otevřených více a museli bychom tedy spojení přihlásit k více workspace a složitě pak komunikaci třídit. Další věcí, která by se komplikovala je, že server by ztratil vědomí o tom, které soubory má uživatel otevřené v případě, že by uživatel přestal na souboru pracovat, ale nechal jej otevřený. Pokud tedy existuje socket, soubor se udržuje v paměti a dříve zmíněná ochranná doba 30 minut se stále posunuje od aktuálního času.

Tento socket má za úkol:

- **Reagovat na změny v obsahu souborů**
  Posílat na server vygenerovaný diff patch(8.13) a distribuovat patch k ostatním stanicím, které jsou ve stejné workspace.

- **Přenášet zprávy z chatu.**
  Stejně jako probíhá broadcasting vygenerovaného diff patch, probíhá broadcasting zpráv v chatu. Chatovací místností je zde workspace s názvem `hash`, tak jak bylo zmíněno již dříve.

- **Inicializovat chat**
  Automaticky po spojení a načtení informací o souboru je klientovi zaslána konverzace v rámci souboru(workspace) za posledních 30 dní.

- **Inicializovat historii**
  Stejně tak, jako inicializace chatu se načítá historie uložených souborů za posledních 30 dní a je zaslána klientovi.

```
io.of('/chat').on('connection', function(socket){
    socket.join('Room1');
    socket.on('message', function(data){
        socket.broadcast.to('Room1').emit('message', 'It works!');
    });
});
```

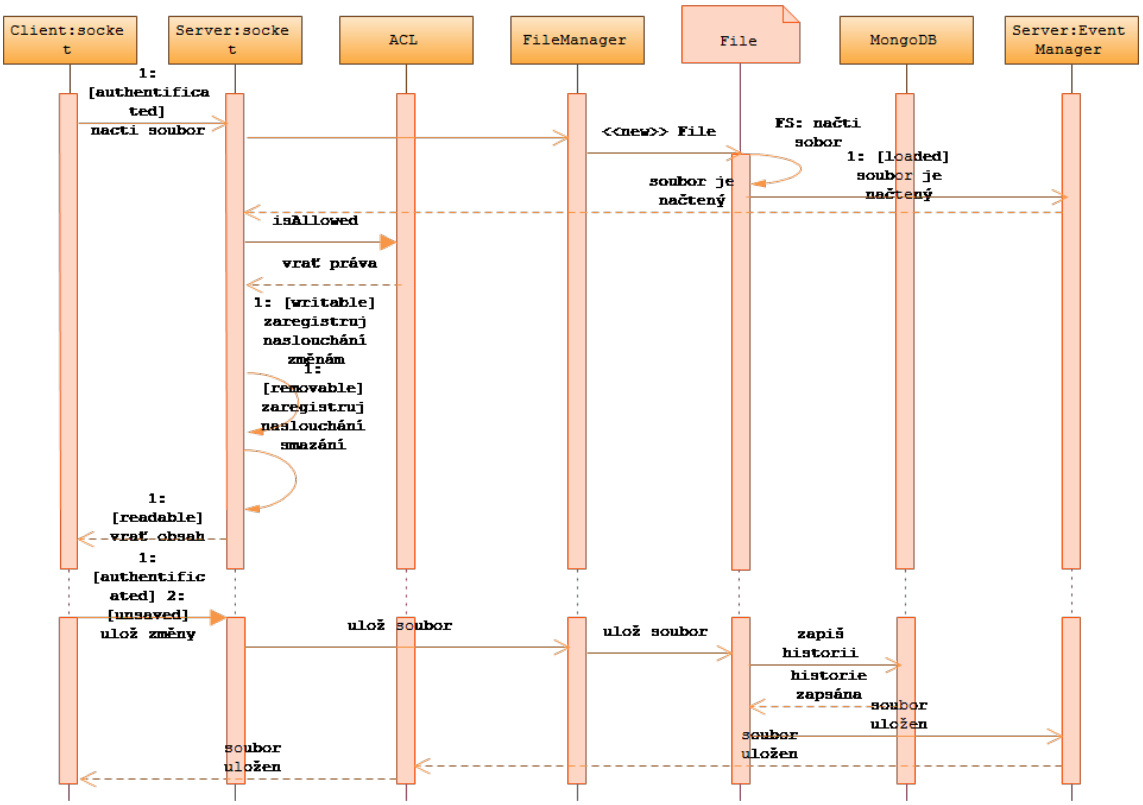
Výpis 18: Socket.io - Join room

V ukázce 18 je práce s workspace. Jsou zde některé věci, které na dřívejších ukázkách nebyly. Každé spojení se dá ještě dále rozdělit podle příchozí URI(`.of('/chat')`). To nám umožňuje, že můžeme v rámci scoketu odeslat více požadavků se stejným názvem,

ale do jiného okruhu. Jakmile socket obdrží požadavek, na kterém naslouchá(zde je to `message`), okamžitě jej odesílá celé workspace. Pokud bychom využili pouze broadcastu bez udání workspace, data by se zaslala všem vytvořeným spojením.

Sekvenční diagram na obrázku 10 zobrazuje proces načtení souboru a následné uložení po změnách. Požadavek na otevření souboru vytváří instanci souboru, ve které je automaticky volána funkce `loadFile` pro načtení obsahu souboru. Jakmile je soubor načten, dává se zpět socketu vědět, že data jsou připravena. Dle ACL se nastaví listenery na operace se souborem, přičemž je předpokladem, že práva k souboru budou alespoň `mode.read == true`. V opačném případě server nezasílá uživateli žádný obsah.

Každé uložení souboru automaticky zapisuje předchozí verzi souboru do databáze, aby byla zajištěna historie. Čeká se na odpověď databáze, že byla data uložena. Poté je serveru oznámeno, že soubor je uložen. Server o tom informuje klienta/klienty(broadcast).



Obrázek 10: Sekvenční diagram - načtení a uložení souboru

8.7 Datová analýza

Podle závěru z datové analýzy bychom měli správně zvolit SŘBD. Databázové úložiště volíme vždy podle

- Typu ukládaných dat.
- Účelu, k jakému uložená data budou sloužit.
- Způsobu zpracování a dalšího využití ukládaných dat.

Rozsáhlé databázové projekty vyžadují robustní SŘBD, které samy nativně obsahují mechanismy pro udržení vnitřní konzistence a zamezují redundancím. Nevýhodou však může být jejich rychlost. Proto při návrhu IS musíme brát v úvahu všechny zmíněné aspekty.

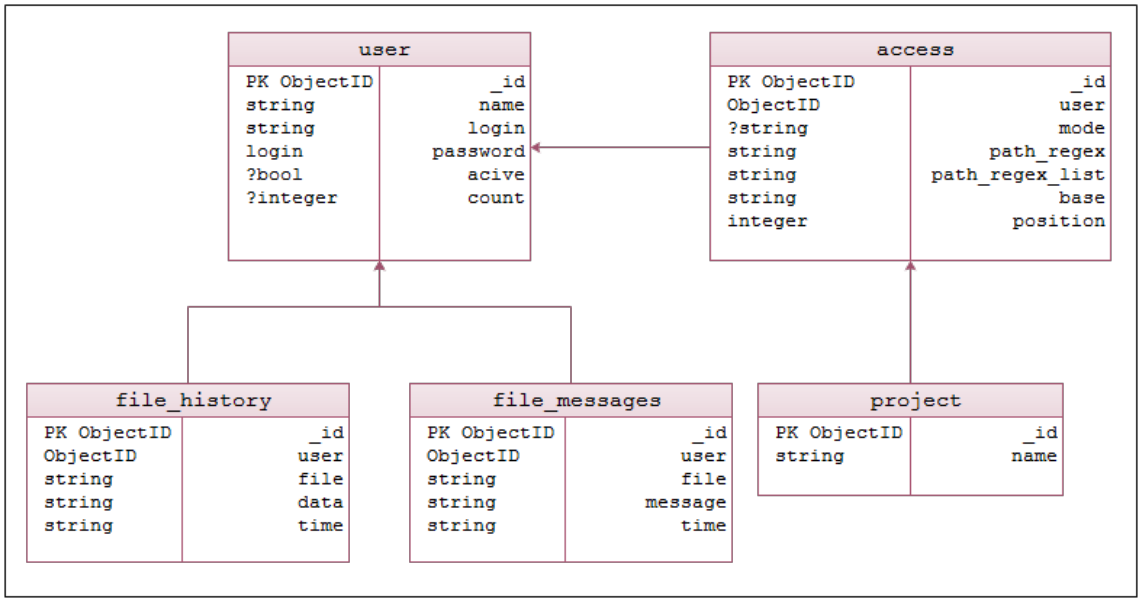
8.7.1 Datový model

- **Data o uživatelích**
  1. Jednoznačný identifikátor uživatele: ID
  2. Jméno uživatele: (string) NAME
  3. Login uživatele: (string) LOGIN
  4. Heslo uživatele v md5: (string) PASSWORD
  5. Příznak, zda je uživatel aktivován: (bool) ACTIVE
  6. Počet neúspěšných pokusů o přihlášení: (int) COUNT
  7. Čas posledního přihlášení jako Unix Timestamp: (int) LAST_LOGIN
- **Projekty**
  1. Jednoznačný identifikátor projektu: ID
  2. Název projektu: (string) NAME
- **Přístupová práva**
  1. Jednoznačný identifikátor souboru: ID
  2. Jednoznačný identifikátor uživatele: USER
  3. Jednoznačný identifikátor projektu: PROJECT
  4. Regulární výraz pro složkovou strukturu: (string) PATH_REGEX
  5. Regulární výraz soubory v koncové složce: (string) PATH_LIST_REGEX
  6. Základ cesty, ve které se bude oprávnění aplikovat: ID
  7. Mód přístupu: (string('r','rw','rwd'))[19] TYPE

[19] Read, ReadWrite, ReadWriteDelete

- **Historii editací souboru**
  1. Jednoznačný identifikátor souboru: ID
  2. Identifikaci souboru ve FS pomocí md5 cesty k souboru: (string) FILE
  3. Obsah souboru: (string) DATA
  4. Jednoznačný identifikátor uživatele: ID
  5. Čas změny jako Unix Timestamp: (int) TIME
- **Data z chatu**
  1. Jednoznačný identifikátor zprávy: ID
  2. Identifikaci souboru ve FS pomocí md5 cesty k souboru: (string) FILE
  3. Obsah zprávy: (string) MESSAGE
  4. Jednoznačný identifikátor uživatele: ID
  5. Čas zprávy jako Unix Timestamp: (int) TIME

Diagram na obrázku 11 předchozí datový model zobrazuje pomocí UML diagramu.



Obrázek 11: UML diagram - ER diagram

8.7.2 Na rychlosti záleží

Datový model aplikace je velice jednoduchý a nevyžaduje robustní databázové řízení. Naopak potřebujeme velice jednoduchý systém ukládaní dat s velmi rychlou odezvou. Tyto možnosti nabízejí nonSQL databáze, které se v poslední době rychle rozvíjejí. Proto

jako SŘBD volím MongoDB5. Tato databáze plně vyhovuje naším požadavkům. Jelikož schéma této databáze není pevně dáno a ukládáme pouze ta data, která databázi předáme, vyhneme se tak ukládání a zpracovávání většího množství sloupců, než ve skutečnosti potřebujeme. Datový model například obsahuje položku s počtem chybných pokusů o přihlášení, ale „nulu“ zapisovat nebudeme, protože ji reprezentuje neexistence této položky, aneb. „žádná informace, taky informace“. Taktéž nebudeme ukládat prázdné nastavení módu ve složce, ve které není očekáván žádný zápis.

Volba je také ovlivněna tím, že nepožadujeme funkci složitých spojování tabulek, jako to nabízí SQL databáze. Vystačíme si pouze s jednoduchými výpisy z databáze podle jednoduchých kritérií. V takovém případě se MongoDB bude chovat jako CacheServer s téměř srovnatelnou rychlostí odpovědí.

8.8 Uploader

Aby aplikace byla plnohodnotným pomocníkem bez nutnosti využívat dalších externích nástrojů, je zapotřebí, aby dokázala nahrávat soubory na server. Byla navržena knihovna, která se stará o uploadování souborů.

Sekvenční diagram na obrázku 12 ukazuje, jakým způsobem uploader souborů funguje. Nejdříve uživatel vyvolá akci, kdy chce uploadovat soubory. To je realizováno pomocí skrytého input elementu typu `file`, na kterém je vyvolána akce `click`. Následně je uživatel vyzván k výběru souborů, které chce na server nahrát. Jakmile uživatel vybere soubory, okamžitě input vyvolává událost `onchange`, na které naslouchá uploader. Uploader vloží vybrané soubory do fronty a začne uploadovat. Smyčka uploadu běží do té doby, dokud server odpovídá, že chce další data(dokud má aplikace spojení se serverem) a dokud fronta není prázdná. Frontu lze plnit dalšími soubory i v průběhu uploadování a není nutné čekat, až se celý proces dokončí.

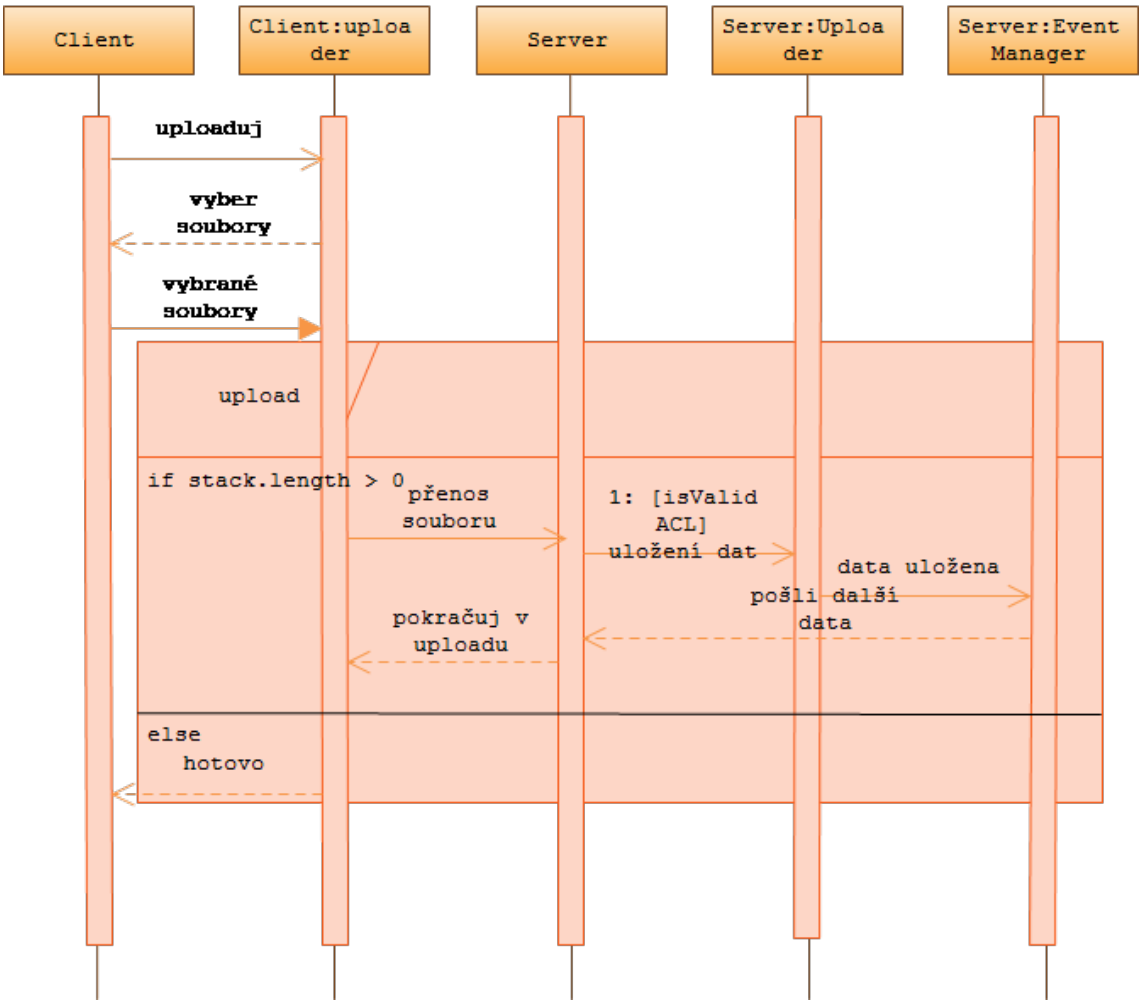
Uploader volá funkci `getFromStack`, která vrací vždy první soubor z fronty. Pomocí funkce HTML5 `FileReader` uploader přečte prvních 524288 bytů a zasílá je spolu s názvem souboru serveru.

Server následně ověří, zda soubor nebyl již dříve uploadován. Tedy zjišťuje, jestli soubor se stejnou `hash` již ve složce existuje.

- **Existuje**
  Existuje-li již takový soubor, tak je následně ověřeno, jestli uploadovaný soubor má stejnou velikost. Pokud ne, je smazán jeho obsah a zapsáno prvních přijatých 524288 bytů. Server dále vysílá požadavek o další data s offsetem zvýšeným o oněch 524288 bytů. Pokud je velikost shodná, je zaslaná klientovi žádost o další data s iformací, od jaké vzdálenosti s uploadem souboru začít.

- **Neexistuje**
  Soubor je vytvořen, je do něj zapsáno prvních 524288 bytů a server vysílá požadavek o další data jako v předešlém případě.

Server společně s offsetem posílá procentuální zastoupení stavu souboru, které je možné využít pro vizuální reprezentaci stavu uploadovacího procesu(dále v kapitole 8.16).



Obrázek 12: Sekvenční diagram - upload

8.9 Autentifikace

Jedním z hlavních požadavků je personalizace dat. Je tedy nezbytné odlišit od sebe jednotlivé uživatele a poskytnout jim správný obsah. Identifikací uživatele se zabývá autentifikace.

Při přístupu do aplikace je jakýkoliv požadavek, kterému vyhovuje nějaké interní routování(routování je blíže popsáno v kapitole 4.4), je automaticky přesměrován na požadavek k autentifikaci. Aplikace nedovolí autentifikaci nijak přeskočit, protože neexistuje případ, kdy by takové chování bylo žádoucí.

Každý uživatel se autentifikuje pomocí unikátního jména(login) a hesla. Tyto údaje jsou ověřovány oproti uloženým datům v databázi a je také nutné, aby uživatel byl aktivní. Být aktivní znamená, mít povoleno přihlášení.

V případě, že uživatel špatně zadá heslo, ale jeho `login` je správný, zvýší se čítač

`count` chybných přístupů o 1. Dosáhne-li tento čítač hodnoty 5, uživatel je automaticky blokuje a již není možné se přihlásit. Hodnota čítače se vynuluje a zároveň se změní hodnota `active` na `false`. Toto vyžaduje zásah administrátora, aby uživatele opět aktivoval a vytvořil uživateli nové přístupové heslo.

Jakmile je uživatelova identita ověřena, vytvoří se v objektu `Auth` nová instance uživatele(pokud již neexistuje) a zapíše se v tomto objektu unikátní `sessionId` tohoto spojení jako reference na instanci uživatele. Na instanci uživatele může existovat mnohonásobná reference. Toho můžeme docílit tím, že pod jedním uživatelem budeme aplikaci používat ve více internetových prohlížečích najednou[20] nebo pomocí anonymních oken. V aplikaci potom konkrétního uživatele získáme funkcí `auth.getUserBySession(sessionId)`, která nám podle zmíněné reference vrátí konkrétního uživatele. Tyto reference velmi úzce souvisí s procesem GC v kapitole 8.11.

Čas vypršení tohoto spojení je nastaven na 30 minut. Na pozadí aplikace se vyvolává v pravidelných půl-minutových intervalech požadavek na znovu prodloužení této doby. To znamená, že k automatickému odhlášení z aplikace dojde po 30 minutách od zavření aplikace. Uživatel se také může odhlásit ručně pomocí tlačítka k tomu sloužícímu(viz 8.16).

Na následujícím výpisu kódu 19 je zkrácená verze procesu autentifikace uživatele. Skutečný kód v aplikaci je o něco složitější.

```
reditor.getMongoDb().collection('user').findOne({
    login      : req.body.login,
    password : reditor.tools().md5(req.body.password),
    active     : 1}, function(err, result){
    if ( reditor.getAuth().getUserById(result._id.toString()) === undefined){
        var user = reditor.getAuth().createUser(result._id.toString());
        user.setData(result);
        reditor.getAuth().addUser(user, session);
    }
    else {
        var user = reditor.getAuth().getUserById(result._id.toString());
        user.setData(result);
        reditor.getAuth().addSession(user, session);
    }
```

Výpis 19: Autentifikace

[20] Jednotlivé prohlížeče mezi sebou nesdílí Site Cookies.

8.10 Autorizace a ACL

ACL je vedle zajištění komunikace skrze WS srdcem celé aplikace. Stará se o personalizaci zobrazovaných dat jednotlivým uživatelům.

V aplikaci se autorizují následující činnosti:

- **HTTP Request**
  Tato autorizace se vztahuje pouze na rotování a tedy na přístupy k jednotlivým podstránkám v systému. Je tedy generování EJS templatu podmíněno tím, jestli je uživatel autentifikován. Pokud není, je směrován na přihlášení. Toto ověření je realizováno pomocí funkce `hasIdentity(sessionId)`. Funkce ověřuje, jestli existuje reference tohoto `sessionId` na nějakého uživatele.

- **Otevření WS** skrze Socket.IO
  Socket.IO(4.5), které zajišťuje komunikaci se serverem skrze WS má svou vlastní autorizaci pro každé WS spojení. Příchozí požadavek v tomto spojení má v hlavičce shodné `sessionId` a můžeme tedy využít ověření identity pomocí již dříve proběhlé autentifikace a ověřit identitu stejným způsobem, jako u `HTTP Request`u. Autorizace v Socket.IO je součástí jeho konfigurace.

```
io.set('authorization', function(req, callback){
    var session = obj.tools().getSessionId(req.headers.cookie)
    if(obj.getAuth().hasIdentity(session) == true){
        return callback(null, true);
    }
    return false;
});
```

Výpis 20: Autorizace v Socket.IO

- **Nastavení módu** oprávnění přístupu k souboru
  ACL má také na starost nastavování módů k přístupu k souboru. V aplikaci používáme 3 základní módy: Read, ReadWrite a ReadWriteDelete. Nejvyšší úrovní oprávnění je tedy oprávnění k mazání souborového systému. Tyto módy se v systémech Unix dají také vyjádřit jako `400` a `600`[21].
  Každá manipulace se souborem nebo souborovým systémem se v ACL ověřuje, jestli ke pro konkrétního uživatele uskutečnitelná. Než budeme pokračovat, podívejme se na následující řádek z databáze, který demonstruje oprávnění k přístupu ke konkrétní složce:

```
{
    "_id"              : ObjectId("53593dcd75d7164e05e256a1"),
    "project"          : ObjectId("5357c75c40c9206cd80fbaf8"),
    "user"             : ObjectId("5357c6fe40c9206cd80fbaf7"),
    "type"             : "rwd",
    "path_regex"       : /^\/home\/www\/stolarstvisvoboda\.cz\/www\/(css|js|img)\/(.*)$/,
    "path_list_regex" :  /^(.(?!\. php))*$/,
```

[21] Unix nerozlišuje zápis, mazání a vytvoření.

```
    "base"              : "/home/www",
}
```

Výpis 21: Oprávnění pro FileSystem

Pořadí vyhodnocení tohoto práva je 3, takže další 2 mu ještě předcházejí. K vysvětlení budeme potřebovat i předešlé dvě, ale již ve zkrácené verzi:

```
{"type" : "", "path_regex" : /^\/home\/www\/stolarstvisvoboda\.cz$/, "path_list_regex" :
    null }
{"type" : "", "path_regex" : /^\/home\/www\/stolarstvisvoboda\.cz\/www$/, "
    path_list_regex" : null }
{"type" : "rwd", "path_regex" : /^\/home\/www\/stolarstvisvoboda\.cz\/www\/(css|js|img)$
    /, "path_list_regex" : /^(.(?!\. php))*$/}
```

Výpis 22: Oprávnění pro FileSystem

1. Server projde veškeré záznamy práv, které jsou k uživateli přiřazeny.
2. Každý záznam obsahuje atribut `base` ve kterém se nachází počáteční cesta ke složce a pro každou cestu spustí podprogram X.
3. Podprogram X znovu prochází v uvedené cestě všechny složky rekurzivně a zároveň na nich testuje jednotlivé záznamy.
   Pokud se jedná o složku, pak je její cesta otestována pomocí `path_regex` a když je přijata, spouští podprogram X sám sebe, ale vstupem je `base` obohacena o podsložku. Složka se zároveň přidává jako další dimenze do pole povolených přístupů[22].
   Pokud se jedná o soubor, pak je složka, ve které leží, otestována pomocí `path_regex`, a samotný název souboru otestován pomocí `path_list_regex`. Je-li soubor přijat, přidává se do pole přístupů ke své složce.

Můžeme si všimnout, že některé záznamy neobsahují `type` nebo `path_list_regex` a nebo ani jeden z těchto parametrů. Jelikož v aplikaci zachováváme stromovou strukturu `FileSystem`u, docílíme tak toho, že u některých složek nižšího zanoření si nepřejeme mít možnost změn ani nahrávání obsahu. Pouze si přejeme, aby server tuto složku schválil jako validní cestu a pokračoval v testování jejího obsahu, který již může být otestován podle některého ze záznamů s již nechybějícími parametry `type` nebo `path_list_regex`.
Takto jsou v instanci uživatele načtena veškerá oprávnění. Následné ověření, jestli uživatel má práva k přístupu k souboru, jaká konkrétní práva má, se řeší jednoduše tak, že se v načtených právech tento soubor vyhledá. Vyhledaná položka obsahuje veškeré informace tak, jako záznam ACL, kterému cesta vyhovovala. To dále poslouží například k operaci `rename`, protože nový název souboru se musí opět otestovat. Následující příklad to demonstruje.

[22] Další dimenze znamená, že pole zachovává stromovou strukturu FS

1. Vytvoříme ve složce `/home/www/stolarstvisvoboda.cz/css/` soubor `test.txt`.
2. Do souboru nahrajeme obsah `<?php phpinfo(); ?>`.
3. Soubor přejmenujeme na `test.php`.

Ačkoliv soubor již nevidíme v souborové složce, protože nevyhovuje žádnému přístupovému právu, ACL nesmí dovolit přejmenovat soubor na nový nevalidní název. Taktéž nám ACL nedovolí soubor přesunout na umístění, které žádnému právu neodpovídá.

8.11 Garbage collector

Jak již zaznělo v kapitole 4.3, NodeJS vytváří aplikace, které běží v jednom procesu. Na rozdíl tedy od aplikací například v PHP, kdy se provede výpočet a po dokončení se celý proces ukončí a načtená data vymažou z paměti, vytvořené objekty uživatele „přežijí". Z tohoto důvodu je nutné implementovat v aplikaci mechanizmy, které zajistí průběžné pročišťování instancí objektů, na které již neexistuje žádná vazba.

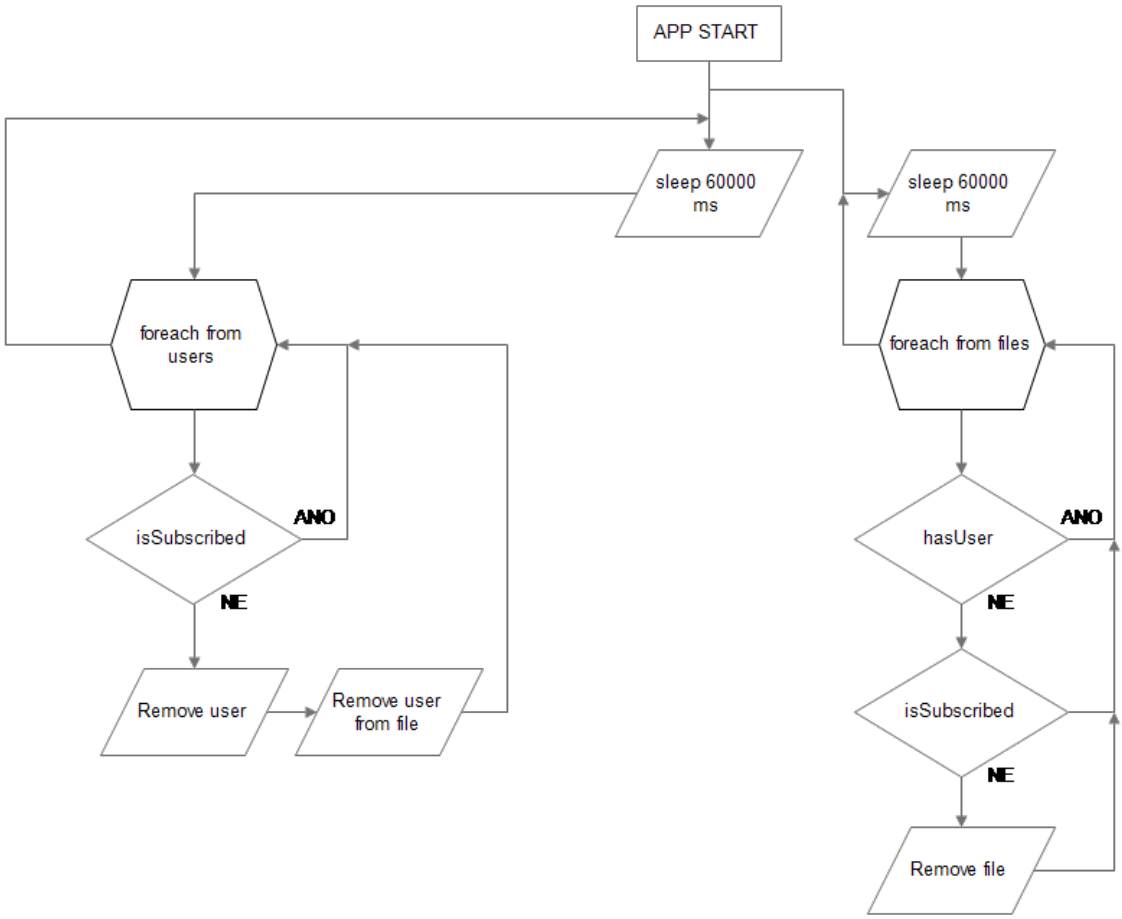
Jedná se zde o instance uživatelů(`User`) a o instance souborů(`File`). Vestavěný garbage collector přímo ve v8 toto za nás nevyřeší, protože maže pouze objekty, na které již fyzicky neexistuje žádná reference. My máme ale uživatele zapsány v objektu `Auth` a soubory v objektu `FileManager`. Reference na objekt je tedy zaručena tím, že tyto objekty o nich vědí a udržují si na ně referenci. O smazání této reference se tedy musíme postarat sami na základě určitých kritérií. Proces garbage collectoru vystihuje následující diagram procesu na obrázku 13.

Jedná se o dva podprogramy běžící v minutových intervalech.

První podprogram Zjišťuje, zda jsou všichni vytvoření uživatelé stále aktivní. Zjišťuje to na základě informace o timestampu unikátních id stanic. Stanice, která se déle jak 30 minut neohlásila, je z registru uživatele vyškrtnuta. Jakmile v některém z dalších cyklů neexistuje žádné podepsané id, je uživatel smazán a zároveň odregistrován ze všech souborů, ke kterým je přihlášen.

Druhý podprogram Prochází veškeré otevřené soubory a zjišťuje, jestli je k nim nějaký uživatel přihlášen. Pokud soubor nemá žádného uživatele, je potencionálně nežádoucí, aby ještě setrvával v operační paměti. Soubor ale může být v ochranné době 30 minut, kdy jsou neuložené změny stále dostupné. Jakmile i tato doba vyprší, je soubor z paměti vyčištěn v některém z dalších cyklů.

Zároveň také každý z těchto podprogramů pročišťuje listenery v `EventManager`u. Všechny listenery, které mají přiřazený workspace související s uživatelem nebo souborem, který byl pomocí GC odstraněn z paměti, jsou taktéž smazány pomocí funkce `DetachByTag`(viz 7).



Obrázek 13: Diagram procesu - Garbage collector

8.12 CodeMirror

Jedním ze základních požadavků na výslednou aplikaci je pokročilejší možnost práce s textem různého typu. Aplikace má vývojářům webů poskytnout přehlednou editaci obsahu a umožnit alespoň základní funkce desktopových aplikací. Rozhodl jsem se z důvodu časové náročnosti tvorby takového editoru využít již existující open-source projekt.

CodeMirror je textový editor implementovaný v JS pro webové prohlížeče. Je navržen konkrétně k úpravě kódu v různých programovacích jazycích. Předně nám jde o práci s `CSS`, `JS`, `HTML` a `Smarty`[26].

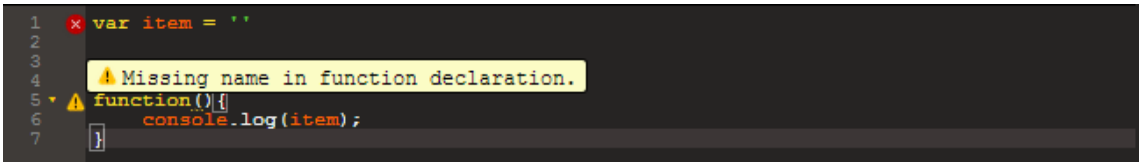
CodeMirror je postaven na principu distribuce úprav do HTML elemtu `Texarea`, kterou zastupuje pomocí zobrazení kódu do HTML. Text, který je vložený v textovém elementu parsuje podle zvoleného módu(programovací jazyk) do HTML podoby podle příslušných zákonitostí, které jsou pro zvolený mód typické. Toto je realizováno pomocí regulárních výrazů a vnořených cyklů. CodeMirror tedy také umí upozorňovat na chyby v kódu, když se nepodaří ověřit některý úsek kódu.

Editor je vytvořen tak, že zachytává pozici kurzoru po kliknutí a na to místo přenese kurzor. Kurzor je však jen imitace opravdového kurzoru pomocí HTML elementu s nastaveným blikáním. Pokud je kurzor vidět, pak to znamená, že celý HTML element editoru je „zaměřen“(:focused). V tomto stavu zachytává veškeré zmáčknuté klávesy listenerem `onkeydown`. Psaní do editoru je tedy rovněž jen imitace. V podstatě tedy text nepíšeme vůbec nikam, jen editor klávesy zachytává a průběžně mění textový element, na který je navázán a aktualizuje zobrazovaný HTML kód. Pozici, kam přidá stisknuté klávesy, je jednoznačně určena pozicí kurzoru. Ke kurzoru se ještě vrátíme v kapitole 8.14.

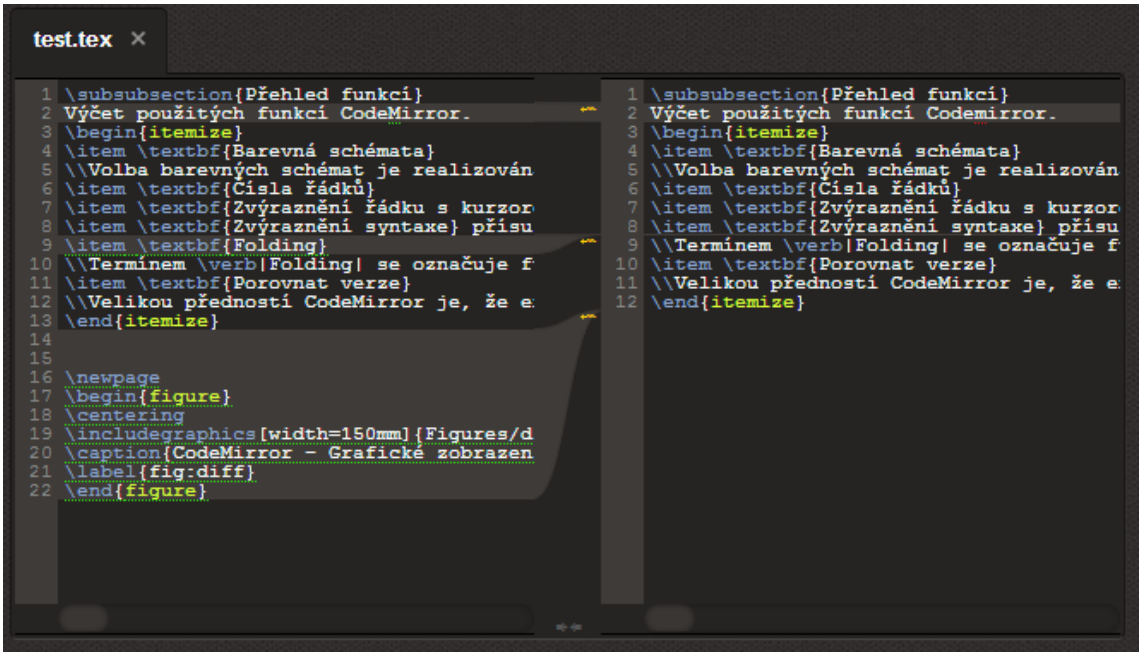
8.12.1 Přehled funkcí

Výčet použitých funkcí CodeMirror.

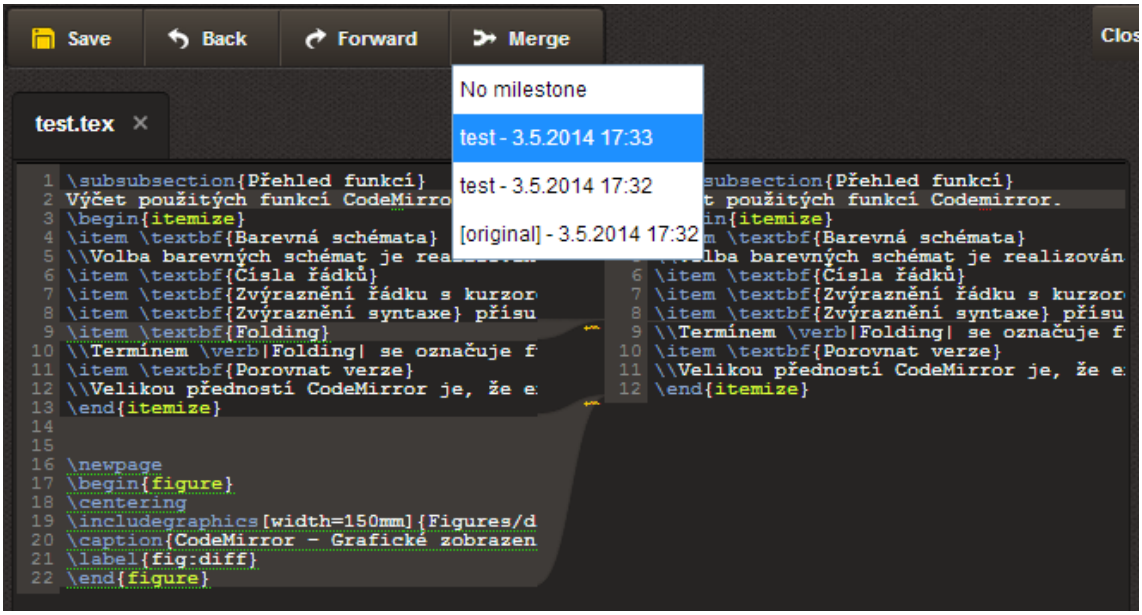
- **Barevná schémata**
  Volba barevných schémat je realizována pomocí načteného CSS souborů s barevnými styly.
- **Čísla řádků**
- **Zvýraznění řádku s kurzorem**
- **Zvýraznění syntaxe** příslušeného programovacích jazyka
- **Kontrola validity** příslušeného programovacího jazyku(na obrázku 14)
- **Folding**
  Termínem `Folding` se označuje funkce editorů, která umí skrývat těla funkcí, polí a jiných souvislých bloků kódu.
- **Porovnat verze**
  Velikou předností CodeMirror je, že existuje doplněk pro grafické zobrazení rozdílů dvou nebo tří dokumentů.
  Tento doplněk taktéž využívá knihovnu `diff_match_patch`6 k výpočtu rozdílů mezi vloženými dokumenty. Tyto změny rozděluje na dílčí úpravy, které graficky znázorňuje, a úpravy je tedy možné parciálně aplikovat viz. obrázek 15.



Obrázek 14: CodeMirror - Nevalidní syntaxe



Obrázek 15: CodeMirror - Grafické zobrazení Diff



Obrázek 16: CodeMirror - Grafické zobrazení Diff č.2

8.13 Diff patch

Teď k samotnému jádru věci. Pro kontinuální práci více uživatelů zároveň na jednom souboru je nutné udržet všechny strany synchronizované a aplikovat veškeré změny jedné stanice na všechny ostatní stanice. Jak bylo řečeno v kapitole 8.12, CodeMirror je navázán na HTML textový element. Na tomto elementu je dále navázán listener `onchange`, který reaguje na jakékoliv změny v tomto elementu. CodeMirror po každé změně upraví obsah tohoto elementu a vyvolá událost `onchange` na tomto elementu. Okamžitě je pomocí `diff_match_patch` vytvořen patch, který převádí předešlou verzi dokumentu na verzi po úpravě. Tento patch je odeslán na server a distribuován všem ostatním stanicím. Na koncových stanicích je patch aplikován. Rychlost, jakou je možné synchronizovat veškeré stanice mezi sebou, je ovlivněna jednak rychlostí a stabilitou internetového připojení a také velikostí vygenerovaného patche[23].

Všechny vygenerované patche se ukládají do instance souboru na serveru a to ze 2 důvodů:

1. Nová stanice, která se k souboru připojí, obdrží obsah souboru před uložením a celý seznam patchů. Tím se dostane do stejného stavu, jako ostatní stanice.

2. Možnost využití funkcí `Undo` a `Redo`. Tato funkce je realizována tak, že se využívá původního načteného obsahu souboru u klienta a na něj se aplikuje list patchů. Server jen posouvá index dopředu nebo dozadu.

8.14 Cursor resolving

Když se více zamyslíme nad synchronizací textu mezi klienty, napadne nás otázka: Co s kurzorem? Je totiž nezbytné zajistit, aby každá aplikace patche buď vůbec pozici kurzoru neovlivnila a nebo jej správně posunula vpřed nebo vzad. Tato, na první pohled, jednoduchá funkce je ovšem dosti problematická, když propojujeme dva naprosto odlišné způsoby reprezentace pozice v textu. CodeMirror totiž uvádí pozici v textu jako `{ch:x, line:y}`. Naproti tomu `diff_match_patch` neudává pozici, kam bude patch aplikován vůbec, protože se pozice vyhledá metodou Fuzzy patch6.2. A poslední komplikací je, že musíme brát v úvahu znaky `\n` jako řádky a správně je odečítat od pozice v kurzoru. Cílem je tedy vše sjednotit.

Následující postup zajistí, že kurzor bude mít vždy správnou polohu:

1. Před aplikací patche si uložíme obsah řádku, na kterém se aktuálně nachází kurzor do proměnné K1.

2. Aplikujeme patch

3. Zjistíme aktuální pozici kurzoru jako délku od začátku. CodeMirror má možnost nechat si vrátit obsah konkrétního řádku. Z pozice kurzoru tedy zjistíme, na kterém řádku se nachází a v cyklu od 0 do `cursor.line-1` sečteme délky vrácených řádků a přičteme jeden znak za každý řádek(`\n` je také znak) a uložíme hodnotu

[23] Zpracování patche pobíhá u klienta, nikoliv na serveru.

do A. Následně přičteme pozici kurzoru na aktuálním řádku `curosr.ch` a uložíme hodnotu do B.

4. Nyní máme veškerá data, která potřebujeme ke správnému posunutí kurzoru.
5. Do proměnné C uložíme `A-B`
6. Do D si uložíme číslo 1, pokud je patch operace insert a -1, pokud jde o delete.
7. Do E si uložíme aktuální řádek kurzoru `cursor.line`.
8. Patch teď metodou `split` rozdělíme podle `\n` a velikost výsledného pole si uložíme do F.
9. Pokud je F větší než 1, pak to znamená, že úprava je víceřádková. Pokud je víceřádková, přičteme k E `D*(F.length-1)`. F.length je velikost pole F a tedy počet řádků v patchi. Uložíme výsledek do C..
10. Nyní máme nastaven kurzor přesně tak, jak byl předtím se správným posunem na řádky. Ale zatím nerespektujeme úpravy, které se dotýkají stejného řádku, na kterém již byl. Bez následujícího kroku tedy přidáním znaku na stejný řádek kurzoru posune text doprava, ale kurzor zůstane na stejné pozici.
11. Teď teprve využijeme K1. Do proměnné K2 si uložíme obsah řádku, na kterém je kurzor po úpravách. Porovnání řetězce `K1.substr(0, A)` a `K2.substr{0, A}`. Pokud jsou shodné, pak na se na stejném řádku před kurzorem nic nezměnilo. Pokud shodné nejsou, pak k hodnotě A přičteme `K1.length-K2.length` a uložíme do A.
12. Kurzor je vyřešen. A tedy uložíme jako `cursor.ch` a E jako `cursor.line`.

Jak je vidět, tak řešení kurzoru není úplně jednoduchá záležitost, pokud jsme nuceni pracovat s různými druhy přístupu. Daleko jednodušší by bylo, kdyby CodeMirror nastavoval kurzor podle přímé pozice, nikoliv pomocí práce s řádky a sloupci.

8.15 Uglify-JS

Uglify-JS je knihovna, která minimalizuje JS kód v několika úrovních.

- **Odstranění** nepoužívaných proměnných a funkcí.
- **Přejmenování** proměnných.
  Proměnné jsou přejmenovány na nejkratší možné délky podle abecedy. Kusy kódu, které neovlivňují dříve deklarované proměnné jsou znovu pojmenovány od začátku abecedy.
- **Přeskládání** funkcí a minimalizace kódu.
  Zápis funkcí a podmínek jsou minimalizovány. Na ukázce 23 je výpis minimalizovaného JS kódu.

```
return i.length > 1 && (e.line += a * (i.length - 1)), o.changedLine[1] = s.changedLine(o.
    content, e), void 0 !== o.changedLine[1] && o.changedLine[0].substr(0, e.ch) != o.changedLine
    [1].substr(0, e.ch) && (e.ch += o.changedLine[1].length - o.changedLine[0].length), e
```

Výpis 23: Uglify-JS minimalizace

Výsledný kód je již těžko čitelný, ale minimalizovaný.

8.16 Prototyp UI

Prototyp uživatelského rozhraní byl navržen tak, aby věrně kopíroval vzhled desktopových aplikací. V režimu fullscreen by tedy měl být nerozeznatelný. V UI jsou také implementovány další ochrany, které se starají o autofocus[24] tak, aby při kliknutí na neklikatelné části, byl automaticky aktivován editor.

Je zabráněno tomu, aby se dal vybírat text mimo oblasti k tomu určené, což by rušilo dojem desktopové aplikace.

8.16.1 Lišta záložek

Lišta záložek slouží k přepínání mezi aktivními a neaktivními otevřenými editory. Uživatel může upravovat více souborů zároveň a může si tak nechávat otevřených více záložek.

V kapitole 8.6 bylo zmíněno, že po obnovení stránky se přenáší seznam otevřených souborů. Podle tohoto seznamu se v postranní liště souborů automaticky rozevřou projekty až k cestě otevřených souborů a vytvoří se automaticky záložky s otevřenými soubory.

Každá záložka obsahuje i tlačítko s křížkem pro zavření souboru. Zavření souboru proběhne bez dotazu na zavření v případě, že jsou všechny změny uloženy nebo k žádným nedošlo.

[24] Automatické zaměření na elementy

8.16.2 Lišta řídících tlačítek

Tato lišta je viditelná pouze, je-li otevřen alespoň jeden soubor. Souvisí totiž s obsahem editovaného souboru. Nachází se zde tlačítka Save, Undo, Redo, Merge a Close.

- **Save**
  Tlačítko Save slouží k uložení změn v dokumentu. Vyvolá zápis do souboru a kontaktuje všechny stanice o tom, že je soubor uložen.
  Po každé změně souboru se tlačítko zvýrazní, aby bylo vidět, že jsou v souboru nějaké neuložené změny. Po uložení toto zvýraznění všem stanicím zmizí.

- **Undo a Redo**
  Tlačítka Undo a Redo slouží k vrácení změn nebo ke znovu provedení změn. Princip funkčnosti těchto tlačítek byl vysvětlen v kapitole 8.6.

- **Merge**
  Tlačítko merge slouží k porovnání předchozích uložených změn v souboru. Po jeho stisknutí se objeví seznam změn za poslední měsíc. Výběrem jedné z těchto možností se zobrazí grafické porovnání verzí viz. obrázek 16.

- **Close**
  Stiskem tlačítka Close dojde k odhlášení se z aplikace a k ukončení programu. Následně je zobrazen požadavek k přihlášení.

V této liště se také nachází ukazatel stavu spojení. Dokud si aplikace udržuje spojení, ikonka zůstává zelená s nápisem ONLINE. Při odpojení se ikonka změní na červenou s nápisem OFFLINE a veškerý obsah dočasně zmizí. Jakmile je spojení znovu navázáno, znovu se vše zobrazí.

8.16.3 Postranní lišty

Postranní lišty lze skrývat a docílit tím širší pracovní plochy pro editaci souborů.

8.16.3.1 Lišta souborů Lišta se stromem souborů je navržena tak, aby umožňovala práci se stromovým výpisem tak, jak jsme zvyklí z desktopových aplikací, včetně vybírání souborů pomocí `Ctrl+MouseLeft`, `Shift+MouseLeft` a kontextové nabídky po pravým kliknutím myši na položku.

Stromová struktura se otevírá kliknutí na obrázek složky, nebo dvojklikem na název složky.

Tato lišta reaguje na změny ve FS a dynamicky se tak mění. Položky se přejmenovávají, mažou nebo se tvoří nové.

8.16.3.2 Chat Tato lišta obsahuje chat mezi uživateli, kteří editují stejný soubor. Nové zprávy se přidávají na konec a dochází k automatickému posunu posuvníku chatovacího okna na konec.

V neaktivních záložkách nová příchozí zpráva přehraje zvuk příchozí zprávy. Byl pro to použit zvuk z ICQ.

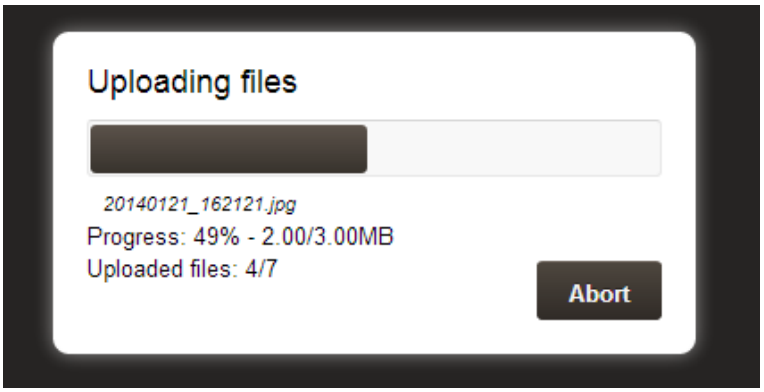
Zprávy se odesílají pomocí tlačítka ve spodní části lišty nebo pomocí `Ctrl+Enter`.

8.16.4 Uploader

Uploader zobrazuje název právě uploadovaného souboru. Vizuální stav nahrávání souboru a textovou reprezentaci stavu. Dále zobrazuje stav fronty a počet již přenesených souborů.

Tlačítko Abort ukončí veškerý přenos a vymaže frontu.

Na obrázku 17 je zachyceno nahrávání souborů na server.



Obrázek 17: Upload souborů

8.17 Testovací verze

Testovací verze aplikace je přístupná na adrese http://93.170.16.189:3000. Všechny dostupné soubory jsou testovací kopie a nehrozí žádné nebezpečí narušení projektu změnou, nebo smazáním těchto souborů.

Login	test
Password	test

Tabulka 2: Přístupy k testovací verzi

Povede-li se někomu nalézt chybu, nebo dokonce celou aplikaci shodit, kontaktujte mě prosím skrze osobní emailovou adresu `martin@lonsky.net`. Taktéž přijímám jakoukoliv konstruktivní kritiku a náměty na další implementaci a zlepšení aplikace.

9 Závěr

V závěru bych rád zmínil, že i přes některé komplikace se mi povedlo aplikaci dotáhnout do konečné fáze a splnit veškeré požadavky, které na aplikaci byly kladeny. Rozhodl jsem se, že aplikaci budu šířit jako open-source. Podle osobního průzkumu to bude první aplikace svého typu, která bude veřejně dostupná. Plánuji zakoupení domény a vytvoření informační a prezentační webové stránky společně s API dokumentací. Také plánuji aplikaci zařadit mezi NPM balíky, aby bylo možné ji jednoduše nainstalovat na veškerých platformách.

Předmětem další diskuse také je verzování souborů. Aplikace umí porovnávat předchozí verze souborů, ale nijak se zatím neřeší větve. Otázkou tedy je, jestli implementovat některý z verzovacích systémů(preferoval bych GIT z důvodů osobních zkušeností a škálovatelnosti). Větvení upravované aplikace by zajistilo tvorby testovacích verzí internetové aplikace, protože momentálně se upravují ostré zdrojové soubory.

V průběhu implementace jsem také narazil na některé problémy se Socket.IO. Konkrétně ve chvíli, kdy jsem ve stejném čase vytvořil několik spojení. Jednalo se o vybrání více souborů ve složce a jejich následné smazání. Bylo to realizováno pomocí nového WS spojení pro každý soubor(viz kapitole 8.6) a skrze vytvořený socket byl zaslán požadavek na smazání souboru. Po smazání byl socket opět uzavřen. Docházelo k problému s handshake. U některých položek nedošlo k handshake okamžitě, ale až po několikerém pokusu, což zapříčinilo poměrně vysokou časovou prodlevu. Tento problém byl částečně odstraněn využitím RedisServeru, ale zcela tento problém nevymizel. Problém se vyřešil až převedením této logiky na řídící socket, aby se pro tuto akci nevytvářely zbytečné sockety. Z toho vyplývá, že s počtem otevřených socketů se to „nesmí přehánět“. Čím méně jich je, tím lépe nebo je třeba zajistit, aby se jich nevytvořilo větší množství v rychlém sledu po sobě.

Musím také dodat, že internetové prohlížeče jsou již připraveny čelit desktopovým aplikacím. V aplikaci využívám výpočetního výkonu klienta a server je pouze prostředníkem při komunikaci. Úzkým hrdlem je však datový přenos, kterému se v aplikacích reálného času nevyhneme. Kvalita internetových služeb se však stále zlepšuje a výhledově toto nepovažuji za překážku.

Tento projekt mě svým způsobem hodně poznamenal a získal jsem zalíbení v aplikacích reálného času a hlavně s prací se serverem jako se stále běžícím procesem, uchovávajícím data stále v paměti. Plánuji se tomuto typu aplikací dále věnovat v navazujícím studiu i mimo něj.

Martin Lonský

10 Reference

[1] Manuel Kissling *The Node Beginner Book,* Leanpub. 2001 ISBN: 1471628442

[2] David Herron *Node Web Development* Packt Publishing. 2011

[3] Kristina Chodorow, Michael Dirolf *MongoDB: The Definitive Guide* O'Reilly Media. 2010

[4] Eugene W. Meyers *An O(ND) Difference Algorithm and Its Variations,* University of Arizona, Tucson, 1986.

[5] `http://nodejs.org` *NodeJS*

[6] `http://goo.gl/vPLQ17` *NodeJS a Vert.x*

[7] `https://github.com/crcn/nexe` *nexe kompilátor*

[8] `https://code.google.com/p/v8/` *Repozitář Google v8*

[9] `http://goo.gl/kaHUDB` *ECMAScript ECMA-262*

[10] `http://jade-lang.com/` *JADE HTML Template engine*

[11] `http://embeddedjs.com/` *EJS Template engine*

[12] `http://i.stack.imgur.com/BTm1H.png` *NodeJS single process*

[13] `https://www.mongodb.org/` *MongoDB*

[14] `http://goo.gl/wzLymW` *Google API for Diff, Match and Patch Library.*

[15] `http://bsonspec.org/` *BSON*

[16] `http://www.w3schools.com/` *w3schools*

[17] `http://goo.gl/PPSNg7` *Mozzila API XMLHttpRequest*

[18] `http://goo.gl/lZYvop` *WebSocket RFC-6455*

[19] `http://socket.io/` *io*

[20] `http://goo.gl/OYAnoc` *ioclients*

[21] `http://ejohn.org/files/ecma-cloud.png` *ECMAScript cloud*

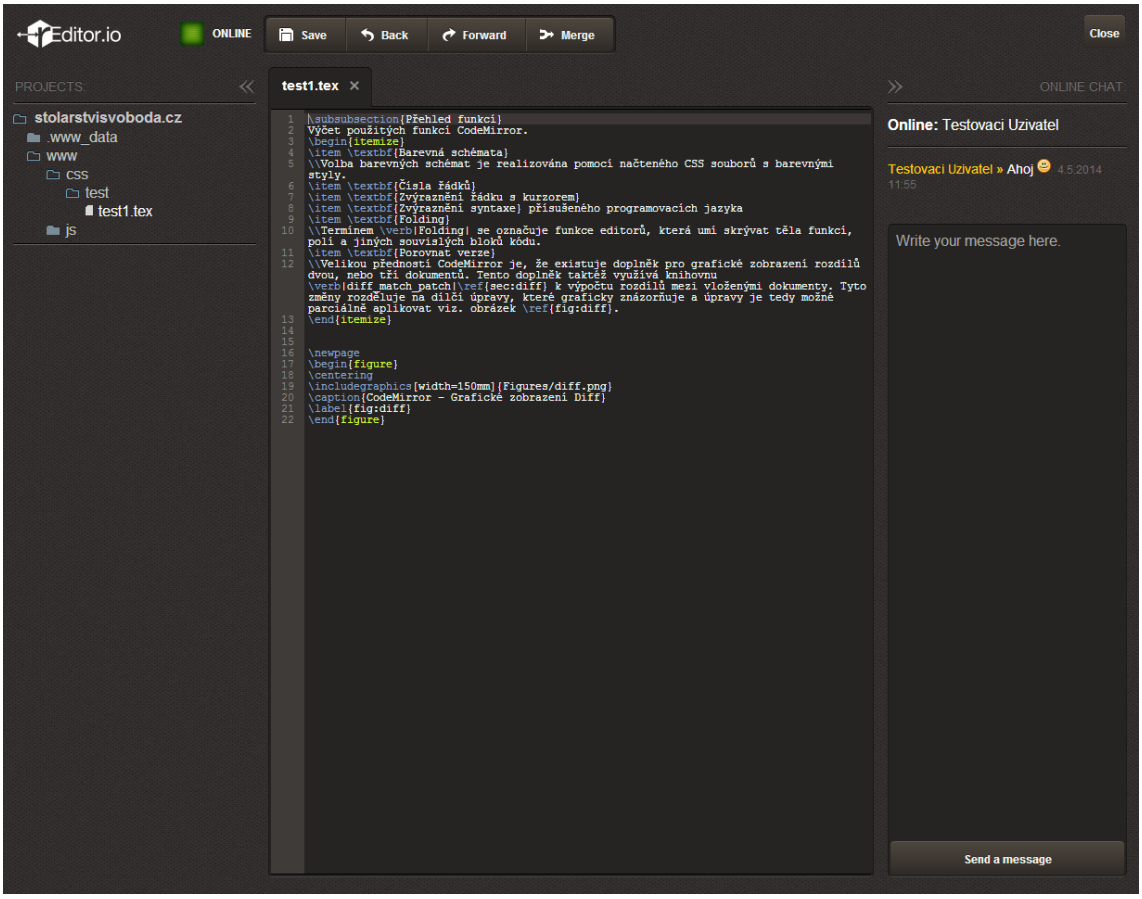
[22] `http://goo.gl/Ha0v1K` *HTML5 formule*

[23] `https://neil.fraser.name/writing/patch/` *Fuzzy patch*

[24] `http://goo.gl/FXIasA` *Hamming distance*

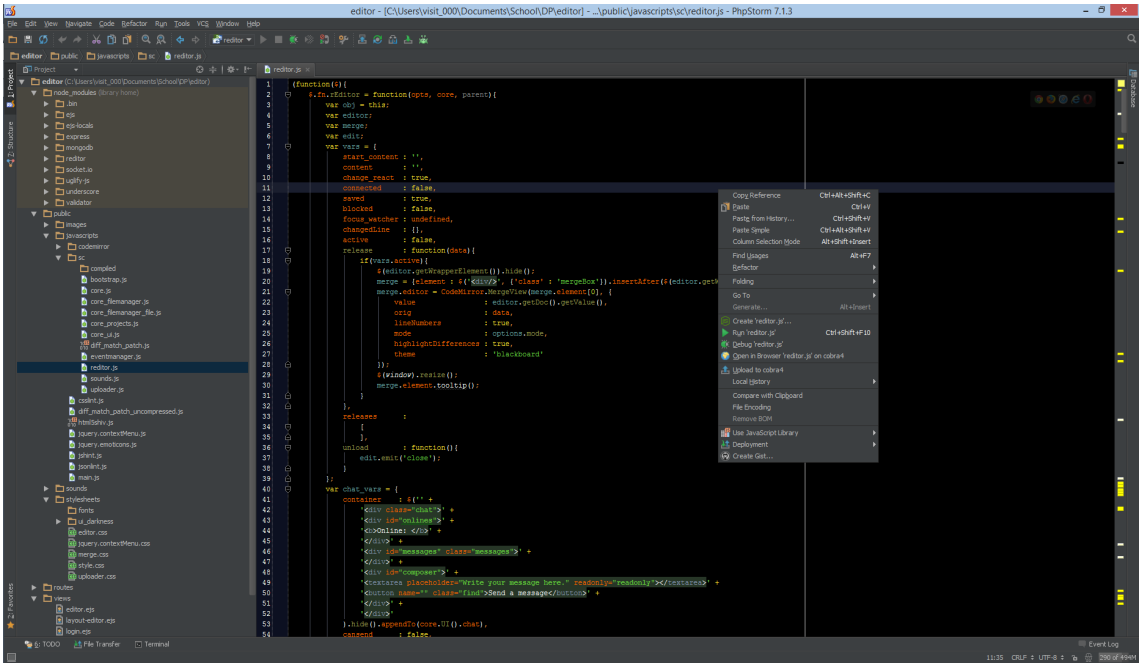
[25] `http://www.levenshtein.net/` *Levenshtein distance*

[26] `http://codemirror.net/` *CodeMirror*

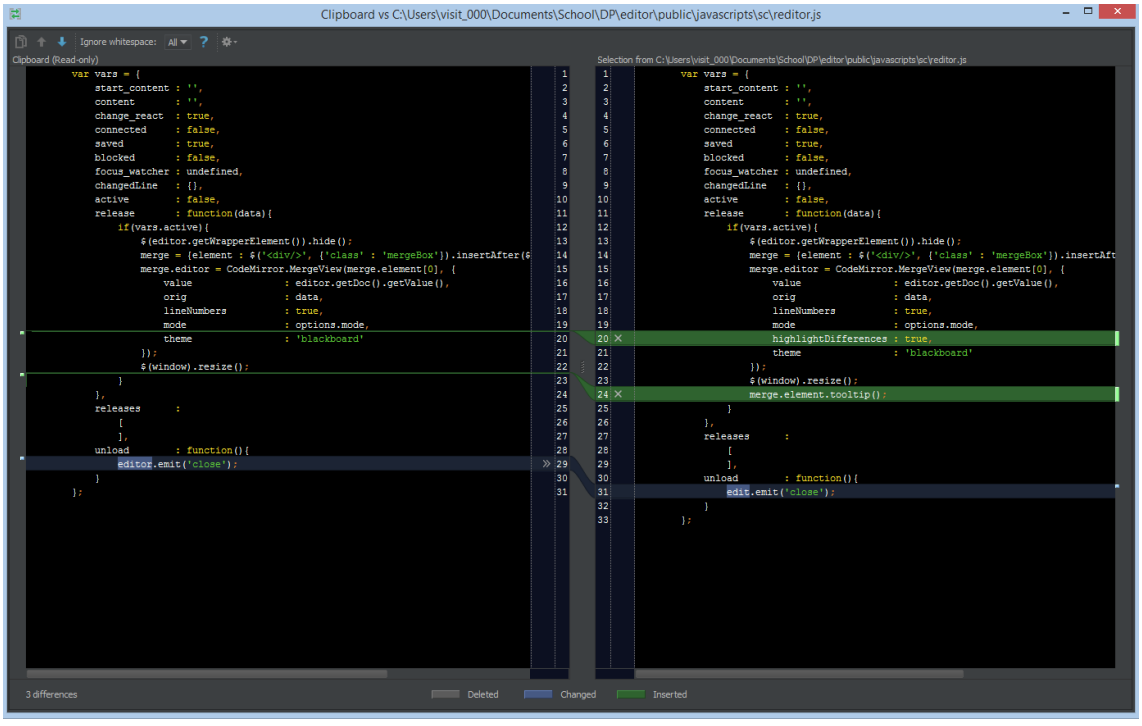


Obrázek 18: User interface

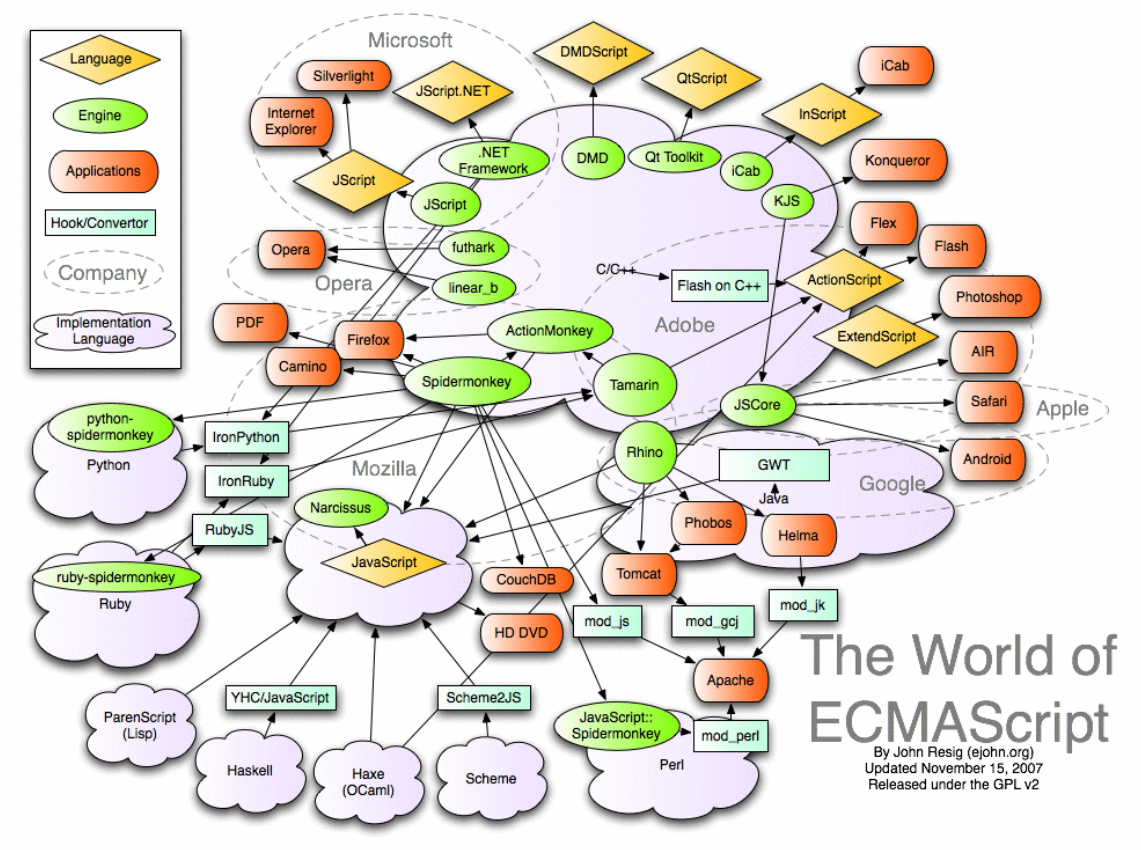
Obrázek 19: User interface - login



Obrázek 20: PhpStorm 7.1.3 - JetBrains s.r.o.



Obrázek 21: PhpStorm 7.1.3 Diff - JetBrains s.r.o.



Obrázek 22: ECMAScript cloud[21]